School of Informatics
University of Edinburgh
Edinburgh, Scotland EH8 9LE

Using Features for Automated Problem Solving

Alex Heneveld

alex@heneveld.org

A Doctoral Thesis in Informatics

May 2007

Abstract.
We motivate and present an architecture for problem solving where an abstraction layer of "features" plays the key role in determining methods to apply. The system is presented in the context of theorem proving with Isabelle, and we demonstrate how this approach to encoding control knowledge is expressively different to other common techniques. We look closely at two areas where the feature layer may offer benefits to theorem proving — semi-automation and learning — and find strong evidence that in these particular domains, the approach shows compelling promise. The system includes a graphical theorem-proving user interface for Eclipse ProofGeneral and is available from the project web page, `http://feasch.heneveld.org`.

Keywords: *problem solving, cognitive science, AI, computer science, planning, CBR, schema abstraction, features, retrieval, analogy, symbolic integration, mathematical reasoning, automated reasoning, Isabelle*

Table of Contents

Part Two: The Expressivity of Features

Part Three: The Benefits of Features

Part Four: The Future of Features

Acknowledgements

Without the unflagging support of a great many individuals and organisations, it would not have been possible for me to undertake this research. I am grateful to the Marshall Scholarship programme for encouraging and funding my study in the UK, and to the University of Edinburgh for providing additional funding for my third year of study. I appreciate the time, wisdom, and challenges that my principal supervisor, Prof Alan Bundy, has shared with me ever since I began the research in 1998. I have also benefited immeasurably from the involvement of other supervisors along the way: Julian Richardson, Corin Gurr, and Paul Schweizer. The feedback of my examiners, Prof Bruno Buchberger and John Lee, was extraordinarily helpful and supportive. Countless other conversations and publications have guided my thinking to this place where I am happy with the product of my research — "countless" meaning, of course, that I could not list them all if I tried. I hope it suffices for me to thank everyone who so contributed, whether they be aware of it or not, and to express particular gratitude to Michael Ramscar, Prof Keith Stenning, and the whole of the Discovery and Reasoning in Mathematics (DReaM) group.

During the long and wee hours spent working for this thesis, I have been sustained by love and joy from my family and friends. I thank every single one of them enormously. Two very special people deserve credit for this sustentation above all: my partner Laura and my son Lucas. This thesis is dedicated to you.

Declaration

As per regulations of the University of Edinburgh, I, Alex Heneveld, attest to the following:

(a) that the thesis has been composed solely by myself;

(b) that the work is entirely my own; and

(c) that the work has not been submitted for any other degree or qualification.

Alex Heneveld

Edinburgh
1 May 2007

Overview

The Possibility of Features

Recent research in cognitive psychology has introduced new theories of human problem solving where features play a key role. In **Part One** of this thesis we motivate and test the possibility of basing a computational system for problem solving on these theories. In subsequent parts of the thesis we explore how this approach differs from other computational techniques, and we identify three major benefits of using this approach for complex problem solving in AI.

We begin in **Chapter 1** by reviewing several cognitive accounts of problem solving. We focus particularly on recent analogical (memory-based) theories (*e.g.*, Sobel, 2004; Gentner *et al.*, 2000) where problem solving is described as a three-step process, using features and schemas:

(1) Features are detected in a problem space.

(2) Features cue problem solving schemas from memory.

(3) Cued schemas are applied to the problem.

A feature is an enrichment to the representation of a problem state, formally defined in our work as "a known label or structured entity which can be associated with a problem state". Crucially, it does not include any information about how to act in that state; this is recorded in schemas, "knowledge elements which describe an action which can be taken in a problem state". This theory is illustrated by examples and experiments from the literature and contrasted with traditional approaches including the production rules theory and its two-step "generate-and-test" process (Newell & Simon 1972).

Following this review of the psychological literature, we give a summary of many important problem solving techniques in artificial intelligence (**Chapter 2**), observing that structural features, given such a prominent role in recent cognitive theories, are rarely used in AI systems[1]. This leads us to formulate our first hypothesis, in **Chapter 3**, which we call "the possibility hypothesis":

(PH) Features can be the foundation of an AI problem solving system.

We have tested this hypothesis by building a general system for problem solving, called Feasch, based on features and schemas[2], with particular support for structural features. To solve a given problem, our system takes a problem state as input and first runs a number of "feature detector

[1] The nearest approximations are the use of "hidden layers" in neural networks and "indexes" in case-based reasoning. We, however, typically use features which have an explicit, recognisable meaning (in contrast to hidden layers) and require that their detection be an integral part of the process (in contrast to CBR, where they are used for optimisation but are not at the heart of the system.

[2] Formal specifications are given in §3.2 in the form of corresponding Java interfaces `IFeature` and `ISchema`, together with abstract implementations, concrete examples, and discussion.

schemas" (akin to step (1) above). Whenever a feature is detected, a "mapping" describes how to modify likelihood values attached to a feature-dependent set of schemas (step (2) above). After many features have been found, the system has a list of candidate "problem action schemas" (schemas which act in the problem state) which are tried on the current problem state (step (3) above), typically in best-first order on the likelihood value. If one is successful, the system repeats the cycle on the new problem state, recursing until either a solution is found or there are no more cued schemas to try locally. Within the system, several execution contexts are described offering different implementational capabilities, including whether features are permitted to cue other types of schemas — such as more specialised feature detection schemas, in which case step (2) leads back to step (1) in the same problem state — and whether mappings are implemented as lists, weights, or more sophisticated dependencies. (The Feasch System and a User's Guide is available from this project's web page, `http://feasch.heneveld.org`.)

Chapter 3 also describes experiments using Feasch on the canonical noughts-and-crosses problem (also known as tic-tac-toe). Valid moves are expressed as schemas, and control knowledge is expressed as features with numerical weight mappings. We confirm (PH) by showing that the system can be used to implement a solution. As yet in the thesis we have made no claims about any benefits of this approach, nor is it even clear whether it is substantially different to other approaches, although this experiment does suggest how features seem to encode control knowledge is ways that appear qualitatively different.

The Expressivity of Features

Part Two explores to what extent the Feasch approach is in fact different to other approaches by applying it to the domain of formalised mathematical reasoning. If features are a substantially different way of expressing control knowledge, we would expect it to have unique performance properties in this domain. Based on the way features seem to encode knowledge, as suggested in Chapter 3, we propose four such potential performance properties, collectively referred to as our "expressivity hypotheses":

- **(EH1)** Features provide reasoned explanations for method selection, *e.g.* for use in generating natural language proofs with causal information.
- **(EH2)** Features enable useful guidance for interactive theorem proving.
- **(EH3)** Features can encode control knowledge for automated theorem proving.
- **(EH4)** Features contribute to modular design of control knowledge.

We argue that the expressive properties described in hypotheses (EH1), (EH2), and (EH4) are generally not true of pre-existing control knowledge techniques used in theorem provers (including production rules, programmatic tactics, and plans); consequently, establishing them for the Feasch System would highlight several ways that the feature-based approach is unique. (In Part Two, our interest is confined to testing this uniqueness; benefits which may be suggested

in the course of this exploration will be explicitly hypothesised and tested more thoroughly in later parts of this thesis.)

Before testing the four expressivity hypotheses we introduce the theorem prover we use, Isabelle, and describe our work integrating it with the Feasch system (**Chapter 4**). The result of this integration is an end-user Isabelle plug-in called the Feature Wizard, including an Isabelle-level "feature description language" and a graphical user interface as shown in the figure below. (This system is also available for download and public use from the project web page, `http://feasch.heneveld.org/`.)

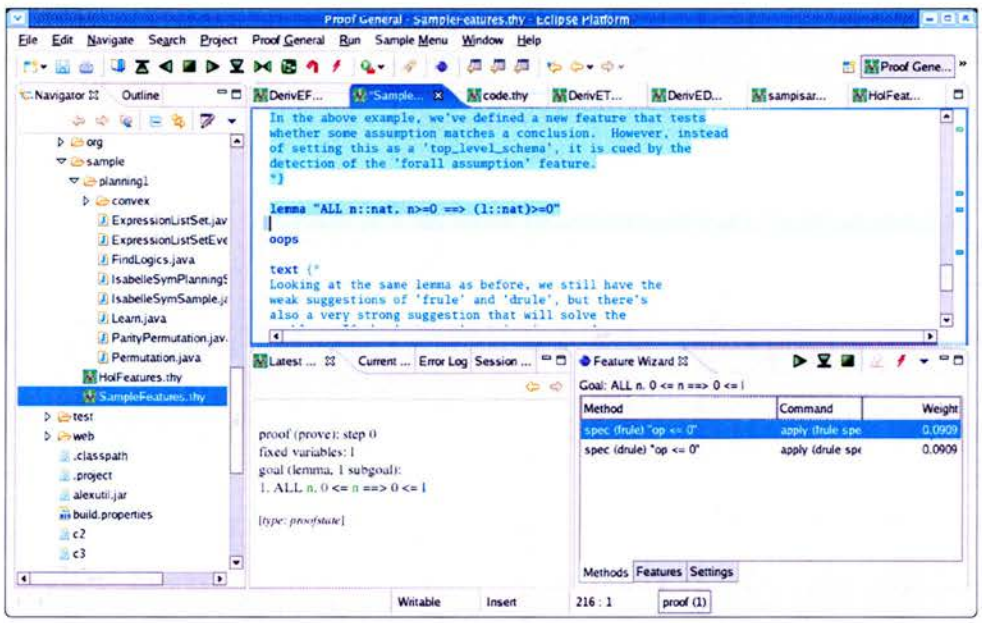

FIGURE: The Feasch-on-Isabelle Feature Wizard

Chapter 5 examines how this tool can be applied to the widely-used Isabelle theory "HOL". We put forward a number of features and method mappings that express the appropriateness of certain HOL rules and methods. We then show how this form of expression allows us to extract programmatically from existing proofs an explanation of why certain methods may have been chosen, and we establish (EH1) by using this information to automatically generate natural language versions of proofs that include causal information (*i.e.* efficient cause, or motivation, in addition to the final cause provided by other systems). The use of the system for guiding interactive theorem proving is then presented, and experiments concerning utility and usability are described. These experiments give strong evidence that the system is helpful to users, confirming (EH2) and distinguishing our approach from other major tools where this type of guidance is not readily available.

Next we turn our attention to the domain of calculus (**Chapter 6**) and attempt to show (EH3) by implementing a decision procedure for evaluating derivatives (of differentiable functions). The experiment is successful in the sense that the system is able to solve differentiation problems, but we find that it is cumbersome to use Feasch to express some types of control

knowledge, particularly those containing multiple sequential steps. This suggests that other techniques are more appropriate where complete decision procedures exist. The benefits of Feasch-on-Isabelle appear to be for those areas where problems cannot be discharged by a straightforward automated program. In these cases, groups of people are likely to have to work together, sharing and building upon each other's control knowledge code. The "black-box" approach of most control knowledge techniques is inappropriate here, for in these instances modularity — including inspectability and reusability — is especially important. We test whether features contribute to modularity (EH4) in part through an experiment where Isabelle users are asked to debug derivative control knowledge expressed in different ways; as subjects are substantially quicker with Feasch than with other ways of encoding control knowledge (even though subjects are *less* familiar with the Feasch approach), the results support the modularity hypothesis.

Of the four expressivity statements, the only one which generally *can* be made for other control knowledge techniques — automation (EH3) — is the only one which was *not* conclusively shown for the Feasch system[3]. This strongly suggests that the feature-based approach is different to other control knowledge representations used in theorem proving, with different attendant strengths and weaknesses, and — it appears — most suited to problem solving in complex domains where decision procedures are not available.

The Benefits of Features

Having established that features yield a *different* approach to problem solving, we turn our attention in **Part Three** to characterising the *benefits* of this approach. As Part Two suggested that it appears suited to complex problems, we explore the challenging task of finding anti-derivatives[4]. **Chapter 7** introduces the implementation of an integral solver in our system and puts forward the first major "benefit hypothesis":

> **(BH1)** The inspectability and modularity properties of feature-based control knowledge are beneficial to multi-agent theorem proving.

We take "multi-agent" to include either human or machine agents because we are hypothesizing *strong integration* benefits of using features — whether one agent can re-use intermediate information in addition to complete results of another agent, irrespective of the animacy of the agents involved. Specifically, the capabilities demonstrated in Part Two are tested in an environment where problem solving is performed in a theorem prover (a machine agent) guided by a user (a human agent) with and without the assistance of the Feature Wizard (another machine

---

[3] Many of these approaches are Turing-complete programming languages, and so their expressive capabilities are theoretically equivalent. In these instances, we say an expressive property is not *generally* true of an approach if the property is not true of how the approach is typically used, if developing the property in the approach requires a significant amount of overhead work, and if this overhead work is tantamount to implementing a different recognised approach.

[4] Although there are complete methods for finding anti-derivatives if they exist (§7.1.1), they are computationally expensive, not inspectable, and not sensitive to boundary conditions — some implementations will even return erroneous solutions. As a result there is interest in inspectable solutions. Previous efforts along these lines (§7.1.1) have had limited success and are used as a baseline for comparing our approach (§7.2.3).

agent); it is shown that the partial automation of our system collaborating with the other two agents yields significantly better performance, as measured both by timings and by percentage of problems solved. We also show by experiment that it is easier for a user to become familiar with a new theory (*i.e.* one written by a different human agent) when control knowledge is expressed using features as opposed to other control knowledge representations. A further suggestion is made (without experimentation at this point) that the causal information that features provide could be used by other machine agents to improve their capabilities.

One such agent is investigated in **Chapter 8**: a *learning* agent which monitors this causal information and attempts to generate improved control knowledge. We look at learning for two reasons. Firstly, in a complex domain where the problem is not always solved (or solved inefficiently), there is scope for improving the control knowledge. Secondly, as features encode control knowledge differently, we might be able to apply learning techniques which have not been successful with other control knowledge representations. Specifically, the introduction of an abstraction layer (the features) between the problem statement and the methods distinguishes three areas where learning could be attempted: automatically computing mappings, automatically finding useful features, and automatically constructing new methods. We propose the second benefit hypothesis:

> **(BH2)** Feasch allows the use of learning mechanisms not applicable to other systems, simplifying the encoding of control knowledge and yielding improved performance.

and we test this hypothesis on the first area of learning, that of optimising mappings between features and methods. (The other two areas are substantially more difficult, and also more applicable to other control knowledge techniques. They are discussed as future work, §9.1.5.) We show that a learning agent in our system is able to find good mappings both for the calculus theory and the general HOL theory, as measured by performance. This demonstrates (BH2) and offers further support for (BH1). From the point of view of a user, this capability means that a developer needs only to define features for a problem domain, and our system can build powerful control knowledge from those definitions.

The Future of Features

Part Four first looks at weaknesses and limitations of the feature-based approach that we have encountered in this research and outlines future work which could be done to overcome them (**Chapter 9**).

In **Chapter 10** we introduce a generic formalism in the spirit of Newell & Simon's classic Generalised Problem Solver (1972), where we use features and schemas to reveal commonalities among many AI techniques. We suggest that this may be a new paradigm for viewing computational problem solving, and we argue three broad claims along the lines of the benefit hypotheses. Whilst some compelling evidence is available for this new framework for problem

solving, we conclude that much more research is necessary to establish it as a decisive improvement on its antecedants; and that this research is worth doing, given the practical advantages demonstrated in Part Three.

Chapter 10 concludes by recapping the seven hypotheses tested in this research and reviewing the ideas which motivate us, the accomplishments we show, and the possibilities we hope to explore. Driven by intriguing recent theories in cognitive science, we set out to invesigate whether features could be used as the foundation of an AI system. Finding that this is possible (PH), we then look at how it might express control knowledge in a different way to other approaches, specifically in the area of formalised mathematical reasoning. Experiments testing four hypotheses (EH*n*) demonstrate that this approach has unique expressive properties: these properties distinguish it from other techniques in the literature and suggest that it is particularly well-suited to complex domains. Focussing on one such domain within theorem proving — evaluating integrals — we are able to establish two major benefits of the feature-based approach:

(1) improved machine-**assisted** and **collaborative** theorem proving

(2) simplified development and improved performance through **learning**

We close by reflecting on how cognitive psychology and artificial intelligence can support each other by testing problem solving hypotheses in different ways: in our research, a recent cognitive theory has lead to a powerful new problem solving system. Our experiments with this system have provided support for it as a computational architecture for theorem proving, and further experiments are planned and described to explore its relevance to other problem domains, both with regards to other AI techniques and to the original psychological theories. We are excited about the prospect of future work in both fields which will continue to develop our understanding of problem solving and the role of features and schemas.

Part One: The Possibility of Features

Chapter 1. The Psychology of Human Problem Solving

Human problem solving has long been a subject of fascination for philosophers, and the various theories developed have affected all our lives, from how we were taught in school to how political and business leaders go about making key decisions. We will begin our account of the subject with two important — and highly contrasting — historical theories, after which we will trace the development of these two traditions through to the state of the art. Following this discussion, we will review the space of possible directions of such cognitive science research, introducing the vocabulary of cognitive modelling, and we will conclude by motivating our own views on an avenue of research which appears fruitful.

Associationism

The study of cognition is often traced back at least as far as Aristotle, who, in *On Memory and Reminiscence* and *On the Soul* (both *c.* 350 BCE), holds the mind to be composed of elements of experience organised by "associations". He gives the example of mnemonic devices, where some piece of knowledge can be reinforced in memory by frequent repetition or, more significantly, by constructing a mental image associating it with other elements. He proceeds to set out four principles by which elements are organised in the mind:

- **Contiguity**: two elements, perceived near each other in time or space, will be linked together in the mind ("associated")
- **Frequency**: the more often an element is perceived (or imagined), the quicker it is to recollect it; and the more often an association is encountered, the stronger it becomes
- **Similarity**: two elements which are similar will be associated, regardless of where and when they are experienced
- **Contrast**: experience of one element may trigger recollection of something completely opposite; or experience of A in association with B may trigger recollection of an experience of A in association with Z, where B and Z are opposite in some sense

The theory of Associationism underwent significant developments during the Enlightenment, beginning with Thomas Hobbes (1588-1679), who posited that complex experiences are merely associations of simple experiences (in *Humaine Nature*, 1650). Hobbes focuses on contiguity ("coherence" or "continguity", in his terminology) as the basis for associations and identifies frequency as the basis of Association's strength. David Hume (1711-1776) reinserts similarity ("resemblance", in his terminology, which has become an accepted standard) as a basis for

the "connexion" of ideas and investigates one more basis, powerful yet elusive (in *An Enquiry Concerning Human Understanding*, 1748):

- **Causality**: two elements, where the occurrence of one often leads to the occurrence of the other, will be associated (Hume notes that this cannot be established absolutely, but can be strongly supported by inductive reasoning on repeated contiguity as well as a number of other factors)

James Mill (1773-1836) extended Hume's approach and added another factor for the strength of associations (*Analysis of the Phenomena of the Human Mind*, 1829):

- **Vividness**: elements and associations will be stronger if the experience where they occurred is intense (also called "liveliness")

Thomas Brown (1778-1820) added two other "secondary laws" for the strength of associations (*Lectures on the Philosophy of the Human Mind*, 1828):

- **Recency**: the more recently that an element is perceived (or considered), the stronger will be its associations; and the more recently an association has been followed, the stronger it will be
- **Duration**: the longer that an element is perceived (or considered), the stronger will be its associations and the more likely it is to be retrieved later

John Stuart Mill (1806-1873), the son of James Mill, continued the study of causality, specifying methods of inductive generalisation — of which the method of difference is preëminent: if A and B cause C, but B does not cause C, then it is likely that A causes C (in *A System of Logic*, 1843). As suggested by this summary of his "logic", his approach has more to do with Associative learning, and less to do with the deductive tradition introduced in the next section, than the title of his book might indicate.

Logic

Starting in the late 18th Century, the Associationist approach, and the Empiricist School which developed it, faced growing challenges from philosophers convinced that there must be more structure to knowledge and reason. Immanuel Kant (1724-1804) rejected the empiricist premise (of Hume and others) that knowledge is necessarily rooted in experience, and sought to develop an *a priori* classification of knowledge (*A Critique of Pure Reason*, 1781), based on rationalism and logic. George Boole (1815-1864) followed the Kantian tradition, seeking to identify "laws of thought" that would be independent of experience. In the introduction to *An Investigation into the Laws of Thought* (1854), he writes:

> The design of the following treatise is to investigate the fundamental laws of the operation of the mind by which reasoning is performed; to give expression to them in the symbolical language of Calculus, and upon this foundation to establish the science of Logic and construct its method; to make that method itself the basis of a general method for the application of the mathematical doctrine of Probability; and, finally, to collect from the various elements of truth brought to view in the course of these inquiries some probable intimations concerning the nature and constitution of the human mind.

Boole develops the syllogistic logic of Aristotle, as well as the work of Chinese, Indian, and Arabic scholars, relating truth-valued propositions as sets and identifying properties of their conjunctions ($\wedge$, $\cap$), disjunctions ($\vee$, $\cup$), and negations ($\neg$). By the end of the thesis, however,

he concludes that his Logic does not account for the breadth of actual human thought. Instead, he argues, it is a normative description of how correct thinking should proceed. His Logic, of course, was fundamental to the development of computers and digital technology; and successive developments, chiefly the work of Gottlob Frege (1848-1925; *Begriffsschrift [Concept Script]*, 1879) and Charles Pierce (1839-1914; "On the Algebra of Logic", 1885) , has led to widely-used current forms of logic, using predicate notation, with formalised universal ($\forall$) and existential quantification ($\exists$) of variables and, in some logics, higher-order predicates and quantifiers.

Predicate logics have become a *de facto* standard of representation, used in a vast range of AI systems (§2) as well as an increasing number of knowledge corpora (due largely to the use of XML and XSL, which are closely related to predicate logic). Our interest in this chapter, however, is in the use of such logics to describe human cognition and problem solving. Although the logical tradition and the associationist tradition have levelled criticisms back and forth — from Kant's critique of Humean empiricism (1781), to Brown's attack on Kantian philosophy in the *Edinburgh Review* (1803), through to present day rivalries discussed in subsequent sections — we do not view them as necessarily incompatible. In the following sections, we use the dichotomy of the two traditions to group recent developments in cognitive science, but we note that the distinction is not always clear, and our choice of where to place theories is at times a subjective one. Our grouping is based on whether a theory is built fundamentally upon logical rules of inference (not necessarily axioms) or built upon associationist principles (which may be incidentally expressed in a logic).

1.1. Memory-based Problem Solving

One of the most well-known examples of associationist principles is Pavlov's theory of behavioural conditioning (1928), where a dog can be trained to associate a bell with food, and will thereafter salivate merely at the sound of a bell. Contiguity of the bell and food is necessary for the association to take place, and the "secondary laws" of frequency and recency can be shown to have a measurable effect on the learning of the association, or at least upon a behavioural manifestation.[5]

At levels above reflex actions, however, and particularly with regards to cognition and problem solving, the interpretation of Associationist principles is unclear. What constitutes an element or an experience? What is the quantitative effect of frequency, or the elusive notion of vividness, on the likelihood of retrieving something from memory? And, crucially, how is the similarity established?

---

[5] This behaviour could be expressed logically, through a statement such as "bell $\rightarrow$ food", with formulas for frequency and recency addressing the strength of this "rule"; Pavlov, however, did not couch his result in this way, and the use of logic is neither explicit nor implicit in the theory, whereas the principle of contiguity is, so we group it in the Associationist tradition.

1.1.1. Gestalt Psychology

By focussing on specific and simplified domains, early 20th Century psychologists in the Gestalt School began to provide answers to some of these questions. Early experiments looked at the perception of shape and interpretations of dot patterns, confirming that the mind forms associations and interpretations based on proximity and similarity, and identifying shape, colour, and size as factors used for judging similarity (the German word *gestalt* means "put together" or "form"). More interestingly, it was observed that where some of these components are the same, the similarity judgment can be invariant of other factors, including symmetries (rotation, reflection, and translation) and lack of "closure". Closure, here, refers to a tendency of the mind to perceive entities in a holistic way, "filling in" details as necessary. "Pragnanz" (a German word meaning precision), is the Gestalt principle that "when things are grasped as wholes, the minimal amount of energy is exerted in thinking." It is by these principles, under Gestalt theory, that people are able to interpret the following images in sophisticated and extremely consistent ways (Wertheimer, 1923):

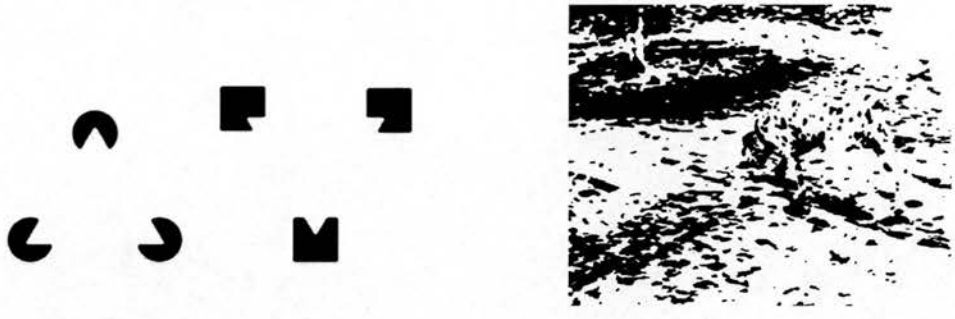

FIGURE 1.1: **Closure in Gestalt Perception.** By proximity, the six small black objects will be perceived together on the left. The principle of closure states that two white triangles will be perceived, and the similarity of shape, colour, and size will associate the two triangles independent of their orientation. The amorphous black blobs on the right hand side are interpreted as a separate, single image, by proximity, and the holistic form of a dog is made out even though none of the dog's constituent parts are individually recognisable.

The importance of holism is championed in Wertheimer's address to the Kant Society (1924), stressing its advance over Associationism of the previous centuries and its advantages over logical approaches:

> For centuries the assumption has prevailed that our world is essentially a summation of elements. For Hume and largely also for Kant the world is like a bundle of fragments, and the dogma of meaningless summations continues to play its part. As for logic, it supplies: concepts, which when rigorously viewed are but sums of properties; classes, which upon closer inspection prove to be mere catchalls; syllogisms, devised by arbitrarily lumping together any two propositions having the character that ... etc. When one considers what a concept is in living thought, what it really means to grasp a conclusion; when one considers what the crucial thing is about a mathematical proof and the concrete interrelationships it involves, one sees that the categories of traditional logic have accomplished nothing in this direction.

Categorisation

Gestalt theory maintains that perception attunes to the essence of an observed reality, at least insofar as it is able to. Perception tends towards "optimal organisation" using associative connections and the "Gestalt qualität" (*ibid.*). The principles of pragnanz and closure, imply that conceptual categories both curtail and inform the act of perception. In other words, people do not view the world as it is, but rather they perceive it through a set of conceptual filters. It is by having a concept of "triangle" and "dog" that people are able to make the potentially more useful interpretations of the images in figure 1.1. As Arnheim succinctly puts it, "eyesight is insight" (1954).

The prospect of perception being any other way is humorously explored in the story of "Funes the Memorious" (Borges, 1944), a man who perceives and remembers every detail, but without any abstract conceptualisation or holistic understanding:

> We, in a glance, perceive three wine glasses on the table; Funes saw all the shoots, clusters, and grapes of the vine. He remembered the shapes of the clouds in the south at dawn on the 30th of April of 1882, and he could compare them in his recollection with the marbled grain in the design of a leather-bound book [...] . These recollections were not simple; each visual image was linked to muscular sensations, thermal sensations, etc. He could reconstruct all his dreams, all his fancies. Two or three times he had reconstructed an entire day. He told me: I have more memories in myself alone than all men have had since the world was a world. And again: My dreams are like your vigils. And again, toward dawn: My memory, sir, is like a garbage disposal.

Borges's account is, sadly, not without its real-world analogue. Luria, in his studies of mnenomism (1975), observed patients with excellent memories but extraordinary difficulties in synthesising them. "Each word calls up images; they collide with one another, and the result is chaos. I can't make anything out of this" (ibid., *cf.* Eysenck 1984).

The process of categorisation is fundamental to cognition, and is inseparable from concept formation and similarity judgments, so unsurprisingly it has been one of the most active areas of cognitive science research. Some of the more influential theories are: feature sets (Katz & Posner, 1964), where categories are defined in terms of features that elements must possess; family resemblance (Wittgenstein, 1953; Rosch & Mervis 1975), where categories are defined as flexible entities based on similarity; prototype and exemplar models (Posner & Keele, 1978; Nosofsky, 1974), where prototypical members define the category; and explanation-based categorisation (Murphy & Medin, 1985; Komatsu, 1992), where categories can be defined by arbitrary theories, and in some accounts affected by contextual situation (Barsalou, 2003).

Insight

With regards to problem solving, we note that in some instances, merely finding the right category is the solution. (Identifying the triangles and dog in figure 1.1 is a trivial example.) It is clear, however, that in many instances, and particularly with complex problems, there is a great deal of cognitive work to be done after perception.

In one classic Gestalt experiment, a chimpanzee is placed in a room with bananas hanging beyond his reach and several crates scattered about the floor (Kohler, 1925). The chimpanzee tries standing on a crate, but still cannot reach the bananas. After several moments of apparent frustration and deliberation, the chimpanzee excitedly rearranges and stacks crates, and climbs them to reach the bananas. Kohler hypothesises that a gestation period occurs, during which the structural essence of a problem is identified, and after some gestation, an "insight" may occur. It is only by understanding the functional parameters of a problem that one achieves crucial insights that enable a solution (Koffka, 1935).

In another experiment, human subjects were placed in a room with a table, a candle, a box of matches, a wooden tray of nails, and a hammer; they were asked to attach the candle to the wall above the table in such a way that it would burn without dripping onto the table (Duncker, 1926). Most subjects failed to solve the problem, for as Duncker concluded, they get stuck on the idea that the tray is there only to hold the nails. Those who solved the problem did so by nailing the tray into the wall. The failure of most subjects to reach this solution Duncker attributed to the principle of "functional fixedness": people's problem solving is restricted by the context in which they are set. Insight is often made possible by overcoming these restrictions.

Further examples of this principle were shown through the nine-dot problem and the pendulum problem. In the first of these, familiar to many schoolchildren, nine dots are given on a piece of paper, in a three-by-three grid, and subjects are asked to draw four straight line segments which pass through all the dots, without retracing lines and without lifting the pen. The elusive solution famously involves "thinking outside the box" (Maier, 1930). In the pendulum problem (Maier, 1931), two strings hanging from the ceiling have to be connected to each other; various objects can be used including a pair of pliers. The strings are too far apart for both to be reached, even using the objects as arm extensions. The solution involves the insight that a string can be made to move as pendulum, overcoming functional fixedness to view the pliers as the pendulum weight.

Transformation and Productive Thinking

Although functional fixedness is a useful concept in explaining obstructions to insight, it does not explain the positive factors that contribute to insight. Maier (*ibid.*) found that subjects were much more receptive to the insight needed for the pendulum problem if he brushed against a string, setting it in motion; this result was true even for subjects who reported no awareness of having witnessed him do this.

Further exploration into the nature of insight was performed with a series of experiments where subjects are asked to think aloud while trying to solve the following radiation problem (Duncker, 1945):

> Suppose you are a doctor faced with a patient who has a malignant tumour in his stomach. It is impossible to operate on the patient, but unless the tumour is destroyed the patient will die. There is a kind of ray that can be used to destroy the tumour. If the rays reach the tumour all at once at a sufficiently high intensity, the tumour will be destroyed. Unfortunately, at this intensity the healthy tissue that the rays pass through on the way to the tumour will also be destroyed. At lower intensities the rays are harmless to healthy tissue, but they will not affect the tumour either. What type of procedure might be used to destroy the tumour with the rays, and at the same time avoid destroying the healthy tissue? (version from Gick & Holyoak, 1980)

Duncker recorded the problem solving methods as subjects recounted them, and found that highlighting the structural constraints was a key step in nearly all the solutions. By restructuring the problem in a way which clarified the issues, subjects were able to find the insight: in the Gestalt theory of "productive thinking", the process of transformation — "looking for structural rather than piecemeal truth" by "realizing structural features and structural requirements" (Wertheimer, 1945) — is a key step in enabling insight. (The solution to the radiation problem is described below, after a visual hint.)

As suggested by the pendulum experiment, however, the use of the speak-aloud protocol entails certain risks. Even assuming honest intentions, people are not necessarily aware of their cognitive processes, and so what is spoken aloud will not tell the whole story. Furthermore, the process of speaking aloud may affect subjects' performance. This was shown in an experiment with the Tower of Hanoi problem, where disks must be moved among pegs subject to given constraints, with interesting results pertaining to problem transformation and transfer (Ewert & Lambert, 1932). Those subjects who spoke aloud while performing the task had markedly worse performance than those who did not, possibly due to the unfamiliar additional burden it entailed or interruption to their concentration. However, when the same subjects were later given a different, related problem, the subjects who had previously verbalised their thoughts performed markedly better than those who had not. The Gestaltist interpretation is that the process of verbalising thoughts provides greater familiarity with structural features, enabling problem transformation that leads to a solution.[6]

[6] More recent experiments have generally validated these early findings, using improved methodology that is not dependent on the speak-aloud protocol (*e.g.* Knoblich and Wartenberg, 1998; and Gick & Holyoak, 1980).

The power of this transformation is highlighted by a more recent experiment using the radiation problem, sometimes with the following diagram and sometimes without (Gick & Holyoak, 1980):

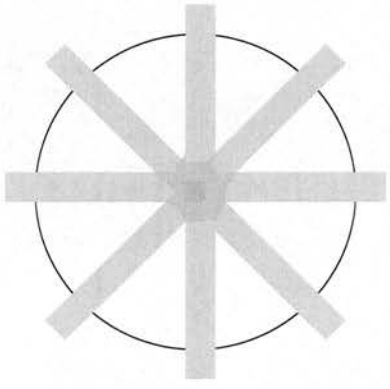

FIGURE 1.2: A Diagram Useful for the Duncker Radiation Problem

By providing the problem transformed into its essentials, the problem is made much easier: subjects given the diagram outperform those without by a wide margin.

1.1.2. Schemas

Productive thinking was initially viewed as an alternative to Associationism, which accounted merely for "reproductive thinking", where a new problem is solved by copying a similar problem with a known solution (Wertheimer, 1945). The distinction between productive and reproductive thinking is not clear cut, however. We have seen that by including a diagram for the radiation problem, by showing the string sway in the pendulum problem, and by encouraging verbalisation in the Tower of Hanoi problem, researchers were able to assist subjects in productive thinking. A recent interpretation of these results is that, in each instance, structural features are cued which support the structural transformation needed to solve the problem. Furthermore, the strength of the effect can be shown to vary in accordance with each of the secondary laws of Associationism, recency, vividness, frequency, and duration (Knoblich & Wartenberg, 1998; Kershaw & Ohlssen, 2001).

In current literature on problem solving, these results are often explained in terms of associative, memory-based schemas. Whilst the term dates back at least as far as Bartlett (1932) and various other closely-related terms have been used (frames by Minsky, in Johnson-Laird & Wason, 1977; scripts by Schank & Abelson, in *ibid.*; and templates by Millikan, 2000) we will start with the definition given by D. E. Rumelhart in "Schemata: the building blocks of cognition" (1980);

> [Schemas are] the fundamental elements upon which all information processing depends. Schema[s] are employed in the process of interpreting sensory data, ... in retrieving information from memory, in organizing actions, in determining goals, ... in allocating resources, and generally in guiding the flow of processing in the system ... . A schema ... is a data structure for representing the generic

> concepts stored in memory. ... [Schemas represent knowledge] about ... objects, situations, events, sequences of events, actions, and sequences of actions. (ellipses and summarisations as quoted from Sabella, 1999)

According to this theory, structural features — such as the swaying string and the vocabulary for the Tower of Hanoi task — correspond to schemas in one's memory, and the spreading activation of these schemas along associative principles constitutes the Gestalt transformation. In "reproductive thinking", the internal representation of a problem activates previously solved example problems or schemas. Where problems require "productive thinking", "Gestalt psychologists ... explained their difficulty in terms of Gestalts, schemas that supposedly organize perceptual information" (Kershaw & Ohlssen, 2001): "the defining characteristic of [these] insight problems is that they activate seemingly relevant prior knowledge which is not, in fact, relevant or helpful" (*ibid.*). The "insight" that is necessary is the activation of schemas which cue helpful prior knowledge, *i.e.*, schemas corresponding to key structural features. Association between certain features in a problem and a relevant schema can be taught (*ibid.*) or strengthened in other ways (Knoblich & Wartenberg, 1998; and others, as discussed earlier), and doing so appears to change difficult "productive thinking" problems into easily solved problems that seem to require only "reproductive thinking".

Experiments studying expertise provide more evidence of the role prior knowledge plays in making problems easier to solve and solved better. De Groot (1946) showed arrangements of pieces on a chess board to expert and amateur players, finding that the experts are significantly better at recalling positions from actual games, but no better than amateurs for chance arrangements. His conclusion was that experts have a larger store of relevant "chunks" — common arrangements of pieces in functional relationships — which enables them to better remember positions from actual games. For random arrangements, de Groot argues, this prior knowledge is completely unhelpful. Experts in chess, it is estimated, have around 50,000 such chunks (Chase & Simon, 1973), of which approximately 7 ± 2 (Miller, 1956) can be active in working memory; or possibly even fewer assuming strong associative links (Halford *et al.*, 1998) . Association can affect the use of chunks, by explicitly cueing appropriate or inappropriate contexts (Eisenstadt & Kareev, in Johnson-Laird & Wason, 1977). The two games Go and Gomoku use identical boards and pieces, although different rules give rise to very different board positions. By telling subjects which game a position was from, their recall ability was improved, and conversely telling them the wrong game gave rise to worse recall performance.

In the Gestalt view, these chunks are examples of "structural features" which contribute to holistic understanding of a problem. In the schema theory view, these chunks are the actual fundamental schemas through which problems are interpreted. It has been argued, however, that this process, interpreting problems through schemas, is in fact a central facet of all cognition: "cognitive features are not added by previous, separate processes; they are expressed in perceptual schemas that are constantly participating in the ongoing activity of perception" (Ben-Zeev, 1988). The "theory of perceptual schemas" (*ibid.*) develops the Gestalt view that "eyesight is insight", recasting it in the modern language of schemas. Expertise, in the schema theory of cognition, is explained by having more schemas, better schemas, and better associative cues for accessing schemas, involved in all aspects of problem solving.

1.1.3. Similarity and Categorisation

The nature of these "better associative cues" merits closer scrutiny. In the schema theory, we can specify that one experience is associated with another if they both activate certain schemas, perceptual or otherwise. The two experiences can be said to be similar with respect to the activated schemas; or as discussed in §1.1.1.i, we could say that they both belong to a common category.

This view is supported by experiments where physics students were asked to partition a set of physics problems into groups. It was found that students who were generally better at physics performed the grouping using structural features of the problem, such that similarly-solved problems were placed together. The less adept students seemed to pay more attention to superficial features unrelated to the solutions. The authors conclude that "[experts'] problem schema were organized around the principles, whereas novices approached problem solving on the basis of the problem's literal surface features". (Chi *et al.*, 1981) Several years later, upon a remark that this work was the single most cited article in the *Cognitive Science* journal, the principal author notes:

> The basic expert-novice result, that experts' knowledge is represented at a "deep" level (how- ever one characterizes "deep"), while novices' knowledge is represented at a more concrete level, has been replicated in many domains, ranging from knowledge possessed by scientists to taxi drivers. This result can also be used to interpret findings in many related cognitive science topics, *e.g.*, analogical reasoning, and concepts and categories. (Chi, 1993)

In a further experiment, Chi *et al.* (1989) found that speaking aloud while studying "solution schemas" can lead to improved solving ability on physics problems. Two reasons for this "self-explanation effect", similar to the findings of Ewert & Lambert (1932) in §1.1.1.iii, are suggested in the literature: speaking aloud leads to a more thorough schema (VanLehn & Jones, 1993) and to a more easily accessible schema (Pirolli & Anderson, 1985).

What this shows is that having detailed schemas, and having them available when given a problem, are central factors that play a role early in the problem solving process. The use of "literal surface features" for classifying problems is consistent with many of the theories of categorisation from §1.1.1.i. However, the fact that complex schemas are involved in experts' grouping of problems gives strong support for explanation-based categorisation, where schemas "encode the typical properties of instances of general categories" (Anderson, 1985) and include definitions based around a theory. Activation of a schema, through perceptual or post-perceptual processes, can suffice to associate a category with a problem. For novices (and in "productive thinking" examples), the schema or category may be merely unhelpful, superficial features, but for experts, the category corresponds to an important concepts in the problem, and the defining schema may directly cue a schematic solution.

1.1.4. Analogy and Retrieval

The Importance of Structure

As noted in §1.1.1.ii, many problems do not have the schematic textbook solutions of college physics, and categorisation — whether on the basis of surface features or structural schemas — does not always lead to a solution directly. Identifying a category (concept) can lead to the activation of a previously solved problem from memory, and the solution might apply analogically to the target problem. In the process of analogical reasoning, mappings are established between the two cases and used to draw inference between them, transferring knowledge from the source to the target in such a way as to provide (one hopes) a solution or insightful structural understanding. Several theories of analogy have been proposed to explain the process (Gentner, 1983; Holyoak & Thagard, 1989a; Hofstadter *et al.*, 1995), using different structural representations (predicate logic, connectionism, and arrays, respectively) and sometimes introducing additional steps (*e.g.*, retrieval, re-representation, evaluation), but nearly all agree that structural similarity is what essentially defines analogical reasoning.

One of the simplest and most influential models for this structural transfer is the theory of structure mapping (Gentner, 1983; implementation described in Falkenheimer *et al.*, 1989). When drawing an analogy, the theory suggests that people seek the best mapping from entities in one case to entities in the other case such that the structural relations between the entities are preserved by the mapping. The fact that "in the solar system, planets revolve around the sun" is thus analogous to "in an atom, electrons revolve around the nucleus"; the structural relationship `in(`M`, revolve-around(`A`, `B`))` is identical under entity matches (*e.g.*, $A \leftrightarrow$ planets $\leftrightarrow$ electrons).

In another experiment involving Duncker's radiation problem (§1.1.1.iii), an analogous story is presented to some subjects, where a general is attacking a fortress on a hill, but if he ascends from any one direction his troops will be wiped out. The solution is also presented: the general has his troops attack from all directions at the same time. Gick & Holyoak (1980) found that whilst only 10% of subjects solved the Duncker problem spontaneously without the story, the rate increased to 40% when subjects were told the story of the general, and increased to 70% when they were specifically told that the story related to the problem.

An interesting observation is that the majority of subjects who were presented the story did not spontaneously find the analogy. The structural transfer is a thought-intensive process, according to most accounts, and as has been shown throughout many of these experiments, the activation of relevant schemas or analogues is particularly important. Other studies show that finding and using analogues from the same domain as a problem — "near transfer" — is much easier than retrieving analogues from other domains — "far transfer" (Gick & Holyoak, 1983). This distinction is reminiscent to the split between "reproductive" and "productive" thinking,

and again association by similarity provides some explanation, since the domain constitutes a schematic category.

Analogical mechanisms apply to abstracted solution schemas in memory just as well as they do to individual previously worked problems (experience). In some instances, such as "textbook solutions", the two cannot be distinguished, and there is little psychological evidence that abstractions are used any differently from experiences — memory experiments in fact suggest the opposite, that we cannot recall events as they were, but only in terms of our schemaas (Schank, 1986). Briefly considering the level of perception and grounded representation, we observe that it would be impossible to have an abstracted schema without prior experiences, and equally impossible to have a meaningful representation of experience without prior abstractions. Although many authors prefer the use of the word "schema" for knowledge that has been abstracted, such a distinction is neither psychologically nor philosophically grounded. As it simplifies discussion to allow the word to include actual experiences (this could be considered "the identity abstraction", as opposed to "a proper abstraction", *cf.* proper subsets), we will follow Rumelhart (1980) in doing so.

Retrieval: Availability and Similarity

The activation of relevant schemas — retrieval — is often considered as a separate process, and the apparent effort involved in analogy certainly suggests the necessity of pre-processing to focus attention within the enormous library of one's entire prior knowledge. Associationism provides some explanation for this retrieval; the importance of vividness, frequency, and recency has been repeatedly shown and, in many instances, quantified (see Squire, 1992, for a good overview); these secondary laws have come to be known collectively as "availability" (Tversky & Kahneman, 1973).

As for the vital element of similarity in retrieval, some much-needed clarification is provided by one series of experiments in particular (Gentner & Landers, 1985; Ratterman & Gentner, 1987). Subjects were told a story of "Karla the Hawk"; two weeks later, each subject was told a "similar" story, which was either superficially similar (a very different story but also involving birds), analogous (but superficially dissimilar, not involving birds), literally similar (analogous and about birds), or unrelated (neither analogous nor about birds). The superficially story involving birds tended to trigger recall of the original story, but the analogous story and the unrelated story did not. However, when shown all four stories, subjects judged the analogous story much more similar to the original than the superficially similar story. This double dissociation, it is argued, shows that superficial features play the prominent role in governing what people retrieve from memory, even if these features may not always be the best cue. A computer simulation following this principle gives a good match to human performance in the story domain (Forbus *et al.*, 1994). Other experiments show that people prefer to use structural similarity to form categories, but identifying structural similarities is difficult (Ramscar & Pain, 1996); and that familiarity with a structural concept facilitates retrieval of a case involving the concept but only if the concept is learned prior to the case (Humphreys *et al.*, 1994).

Abstraction

How do people acquire these concepts? Many accounts for "schema formation" have been given in the literature, and a common view is that they are abstracted from commonalities across experiences (Gick & Holyoak, 1983). By preface, we note that a schematically-defined category need not necessarily be invoked directly in order to perform an analogy. If the target problem shares enough known features with some source case in memory, that might be enough to retrieve the source even if the deeper common structure was not observed or remarkable beforehand. This was found, for example, when students were given particularly difficult physics problems and it was observed they preferred to follow well-defined examples (VanLehn, 1996). Once a source analogue is found, the thought-intensive process of transferring the solution (such as through structure mapping) can lead to an abstraction of the common principle; this abstracted general representation can be stored as a schema. This process of abstraction has been intensively studied, from early principles of induction (Mill, 1829) through to causal learning (Sobel, 2004) and explanation-based generalisation (Schank, 1986).

Some theories posit that the abstractions may not even be stored or made explicitly. According to exemplar models of categorisation (Nosofsky, 1984), a family of similar elements can be constructed without a defining schema being made explicitly, as evidenced by humans' use of categories without names and without vocalisable definitions. The entire analogy-making process, some argue, is merely a part of "high-level perception" (Chalmers *et al.*, 1998), and all the illustrations are simplified approximations. Others find the question of whether feature-perception and schema-representation are explicit to be a moot point:

> I suspect that the argument about whether concepts and knowledge are really perceptual or symbolic will turn out not to be a substantive issue, but instead an argument about what should be called "perceptual". (Murphy, 2002)

Even if these entities are not necessarily explicit in the mind, the fact that people share a vocabulary of concepts — and the speak-aloud experiments demonstrating the self-explanation effect — suggest that they are at least natural artefacts of communication. In models trying to explain cognition, treating perception as part of the process and making features and schemas explicit, we feel, greatly extends the inspectability of the process even if they are approximations to a much more complicated neural phenomenon.

What unites the theories described in this section — beyond details such as whether abstraction is explicit or implicit and whether problem solving occurs through perception, categorisation, or analogy — is the view that all problem solving is shaped by prior experience. Features in a problem are identified and these features cue schemas from memory. The cued schemas may be known textbook solutions to physics problems, which apply directly to the problem, or analogous stories which suggest inferences useful for the problem. The process might not be explicit, and the solver might have no awareness of any concept to explain the cues. The process might iterate over and over, and of course the process might give poor results, at times. Applying prior knowledge to a new domain can be difficult. However, the capability of retrieving relevant knowledge, from a vast store of acquired schemas and experiences, is a much more difficult task which the mind does effortlessly.

1.2. Logical Accounts

Another common facet of the memory-based theories just described is that the knowledge *is* the problem solving mechanism. Perception, categorisation, analogy, and learning are all explained in terms of the existing contents of one's memory. Logical accounts argue against this *tabula rasa* of the Empiricists, maintaining that a formal mechanism for thought is hard-wired in the brain — or at least that such an abstract mechanism is the best high-level approximation. Prior knowledge affects this process only insofar as the logical mechanism uses the knowledge or (in "production rule" formulations) when it is abstracted enough to become a general rule in a logical form.

In developmental psychology, Piaget famously identifies "formal operations" as the characteristic of the fourth and final stage in child development (1983). The challenge of determining the logic of these formal operations, continuing Boole's original goal (§1.ii), was taken up by Jaskowski (1934) and Gentzen (1935). It is the latter's system of natural deduction, based on 13 rules of "introduction and elimination", which has been particularly important:

> Ich wollte zunächst einmal einen Formalismus aufstellen, der dem wirklichen Schlieen möglichst nahe kommt. So ergab sich ein „Kalkül des natürlichen Schließens". (First I wished to construct a formalism that comes as close as possible to actual reasoning. Thus arose a "calculus of natural deduction".) (*ibid.*; translation from Wikipedia, "natural deduction")

Its import, however, has been primarily among logicians, just as with Boole, because even if it is not a descriptive model of human thought, this style of natural deduction is regarded as relatively easy to understand.[7]

1.2.1. Logics for Non-logicality: The Wason Selection Task

One reason for the divergence between strictly logical and psychological accounts has been the repeated confirmation that untutored subjects perform very badly on logic problems. Classic evidence for this is given by the Wason Selection Task (1971), where subjects are shown four cards:

A K 4 7

Subjects are told that each card has a letter on one side and a number on the other, and their task is to turn over as few cards as possible to determine whether the statement "if a card has a vowel on one side, it has an even number on the other" is true or false. The expected answers — A and 7 — are given by less than 10% of subjects in typical experiments (*ibid.*).

Proponents of the logical approaches, however, maintain that people's systematic errors can be predicted by logical rules. The modus tollens rule allows inferring $\neg P$ from $P \rightarrow Q$ and $\neg Q$; in the Wason Selection Task, it means a card showing an odd number must have a consonant overleaf, and corresponds to the most commonly overlooked violation of the rule. Various models

[7] It has, however, fallen out of favour among many contemporary logicians because it is not convenient for meta-logical reasoning.

suggest that people prefer other inference rules to modus tollens or that it is not a primitive rule but rather a lengthy chain of other rules (Braine, 1978; Rips, 1994). Errors and human response times in logic problems can be correlated with the use of non-preferred rules or with the length of the inference chains required to solve them; computer models using sets of primitive rules can give a good fit to human performance data (*ibid.*).

A very different computational approach to explaining the data is the theory that people construct mental models (Johnson-Laird & Wason, 1977) enumerating certain possibilities in certain rule-definable ways; these enumerated possibilities frequently exclude the modus tollens case. Yet other theories postulate that people interpret the problem with varying semantics, such as a biconditional "if" (cards have both a vowel and an even number or neither; Wason, 1968) or deontic rules (whether a rule has been violated, as opposed to its truth value; Geis and Zwicky, 1971; Cosmides, 1989), correctly using consistent logics appropriate to those semantics.

1.2.2. Production Rules

One of the most influential theories of human problem solving takes a related view of cognition, hypothesising that instead of completely abstract natural deduction rules, real-world knowledge is "compiled" into procedural production rules expressed in the logical form "`if <precondition> then <action>`". Logic Theorist (Newell & Simon, 1956), an early computer program discussed more in §2.3, used "transformation rules for logic", expressed as productions, to automatically prove theorems in formal logic. Whilst LT was one of the first inspiring successes of artificial intelligence, its behaviour differed quite markedly from human subjects'; extensive observation of speak-aloud protocols led to the development of a novel system, General Problem Solver (Ernst & Newell, 1969), and an accompanying theory (and groundbreaking book) of *Human Problem Solving* (Newell & Simon, 1972).

This research, in its early days, looked at finitely-specified domains where little or no outside knowledge need to be brought to bear on the problem — including the Tower of Hanoi problem. Whilst such a condition may seem severely restricted in light of the memory-based theories described in §1.1, it had the advantage of simplifying the issue to an extent that striking commonalities could be concluded across many problem domains:

- Problem solving is a search for a "goal" solution within a space of states possible in a problem domain.
- Production rules describe the operators for valid state transformations.
- Production rules may also encapsulate procedural knowledge about which productions are appropriate to the current goal.
- Heuristic strategies guide the selection of productions and the search process.

Using production rules, Newell & Simon are able to give a close account of human performance in a variety of tasks, including crypt-arithmetic, water jug problems, and river-crossing with constraints (*ibid.*). These models, however, frequently relied on the use of strategies, such as hill-climbing, means-end analysis, or best-first search (§1.3, §2), and these strategies are not accounted for within the theory itself. The productions and the strategies had to be hand-coded in the system, with domain-specific productions necessary for each problem type such that solutions resembled decision procedures written in any programming language. Crucially, the theory did not explain where the production rules or the strategies came from, and it did not provide any explanation for problem solving in knowledge-rich domains.

1.2.3. Cognitive Architectures: Soar and ACT-R

Since this seminal work, production rule systems have been extended with the aim of providing "cognitive architectures", a "fixed system of mechanisms that underlies and produces cognitive behaviour" (Newell *et al.*, in Posner, 1989). Two major examples of this approach are Soar (Laird *et al.*, 1986) and ACT-R (Anderson, 1985). In both architectures, prior knowledge is stored in associatively-cued memory (as relatively fine-grained elements, in contrast to schemas; Newell *et al.*, in Posner, 1989), and in both architectures, productions guide behaviour to achieve specified goals.

The most basic difference between the two is that Soar maintains all knowledge as productions, whereas ACT-R maintains distinct "declarative" and "production" memories. The developers of ACT-R base the distinction on a commonly accepted dichotomy in neurophysiology, where declarative ("memory that stores facts and events ... textbook learning", Wikipedia) and procedural ("implicit memory ... of skills and procedures", *ibid.*) memories are dissociable (there are cases of brain damage where one functions without the other, and *vice versa*) and localisable (different brain regions are involved for each). This dichotomy, however, usually places procedural memory as a lower cognitive sub-system (*e.g.*, learning motor skills) having very little to do with conscious problem solving. The distinction in ACT-R seems to correspond more closely to a split within declarative memory between "episodic" and "semantic"; this split is neither dissociable nor localisable, and there is much less consensus on their clear separation.

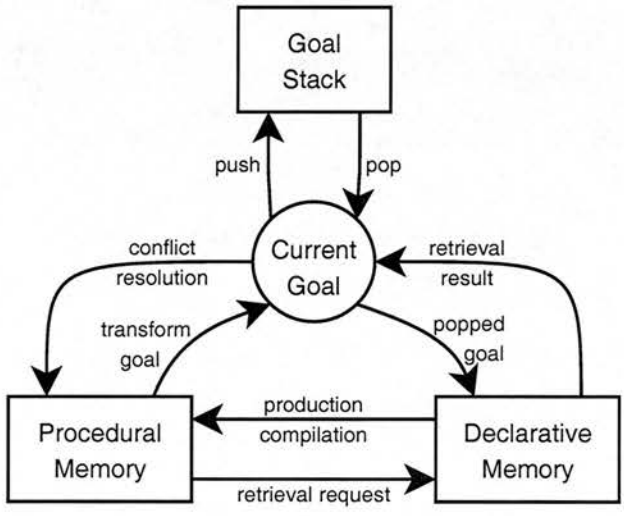

FIGURE 1.3: Problem Solving in ACT-R (adapted from Anderson, 1995)

A second difference between the two architectures is in their mechanisms governing the selection of production rules. ACT-R uses a "conflict resolution" phase to select a single production, by weighing rules according to "expected effort and expected success" (Anderson & Schunn, 2000), parametrised formulas based on past success and the time taken to verify (and instantiate) preconditions. In Soar, all associatively cued productions fire, affecting what are effectively compartmentalised problem representations; a subgoal is created to select among the productions, using a production subsystem to compare the results and identify a unique continuation. Subgoaling is also the principal learning mechanism of Soar:

> All long-term knowledge in Soar arises through chunking, a learning mechanism that compiles the results of subgoal search into a production rule that can produce the result without subgoaling. This implies that Soar can only acquire long-term declarative memory by learning a rule that encodes the appropriate conditions in which the declarative memory will be needed. (Johnson, 1997)

In ACT-R, there are a number of learning mechanisms. Conflict resolution entails some "passive learning", as does the addition of new experiences to declarative memory. The crucial step, however, is the transfer from declarative memory to procedural memory, described recently as follows:

> With respect to procedural knowledge, production rules are learned in ACT-R by a process we call analogy. For analogy to work in the ACT-R theory two things have to happen. First, the ACT-R must come upon a situation where it wants to solve a goal. ... Second, the learner needs an example of the solution of such a goal. ... In this situation, the ACT-R analogy mechanism will try to abstract the principle in the example and form a production rule embodying this principle which can then be applied in the current situation. Once formed this production rule is then available to apply in other situations. (Anderson & Schunn, 2000)

The description is very similar to schema theories of analogy; in practise, however, the majority of models rely on four simpler means of forming productions:

- **Proceduralization**. If a rule accesses declarative memory, and uses the knowledge from declarative memory in its action, then learn a new rule that is identical to the old rule but has the retrieved knowledge instantiated into the rule, eliminating the declarative retrieval.
- **Composition**. Collapse a sequence of rules into a single rule that performs all the actions of the individual rules of the sequence.
- **Generalization**. If there are two rules that are similar, create a generalized version of these rules. This can be done by removing conditions or by substituting constants by variables.
- **Discrimination**. If a rule is successful in one situation but not in another, add conditions to the rule to make a more restrictive version that only applies in successful situations. (Niels & Taatgen, in Nadel, 2000)

These mechanisms are all implemented as internals to ACT-R; they are not described by production rules and are not "learnable" within the theory. Nevertheless, the number of tasks that have been modelled by productions in ACT-R is impressive, and nearly all models are able to give a very close fit to data on actual human performance in problem solving tasks (Anderson & Lebiere, 1998).

It is worth noting that analogy mechanisms have also been implemented as explicit ACT-R models, using inspectable production rules (Salvucci & Anderson, in Anderson & Lebiere, 1998),

and providing one of the only illustrations of "far transfer" in a production architecture (Murdock *et al.*, 1998). In these instances, the production rules are hand-coded and no attempt is made to show either that they lead to the learning of productions or that the analogical productions themselves can be learned. The same is true for means-end analysis and other heuristics identified in §1.2.2 and §1.3.

In general, "different strategies are simply represented with different productions (or sets of productions)" (Anderson & Schunn, 2000), which are hand-coded; or they constitute an internal part of the underlying cognitive architecture. Both ACT-R and Soar develop the theory of cognition as using production rules by specifying how this can be done more fluidly than in the original statement (§1.2.2). The theories have been attacked for having too many tunable parameters in their implementations (latency times can — and frequently are — indicated for every single production in ACT-R, leading some to wonder if the degrees of freedom would permit modelling of nearly any behaviour), but there is little question that they have greatly increased support for the logic-based view that "`if ... then`" productions are the best basis for modelling cognition.[8] They have also provided computational specifications for how the memory-based doctrine of associationism could work[9].

1.3. Strategies for Problem Solving

Whether they are implemented as schemas, productions, or internal control knowledge, nearly all of the theories presented here insist on the importance of certain common strategies for problem solving. Newell & Simon focus on search, with various best-first heuristics; this technique has been fundamental in machine problem solving (see §2.1.2), but evidence suggests people's capacity for search is limited to small state spaces (*e.g.*, the children's game of "noughts and crosses", §3.3) or to a small "lookahead" portion of large state spaces.

The literature on attempts to teach problem solving is one of the richest sources of identifying practical strategies for problem solving. These accounts have mainly been anecdotally acquired by educators with relatively little interest in proving the actual use of these strategies (or in identifying their origins). Though that may be, their effectiveness is undisputed. The strategies — and the books — have proven popular among students as well as everyone from business coaches to AI researchers (Mayer, 1983), and experimental evidence shows that they can, in some instances, improve problem solving ability (§1.3.ii).

---

[8] As in §1.2.1, productions need not correspond to logically valid connectives (although they normally do) and may bear no resemblance to "natural deduction" systems. Issues of soundness and completeness are fundamental in logic and are important from a standpoint of complexity, but have been largely ignored in cognitive models after Logic Theorist (and it is debatable whether LT is a cognitive model, even if it was inspired by and led to principles of human cognition). There is, however, no reason to expect these issues to be applicable to human thought. The essential character of production rules is their form as boolean `precondition` → `action` propositions; systems of these rules are normally hand-crafted for certain domains, in the same way that natural deduction and LT were crafted for the domain of propositional logic, and the use of long-term memory is minimal in most models, so we view production architectures as primarily continuing the "logical" tradition.

[9] Such an account is otherwise severely lacking in the schema theory literature. This is but one of many ways that the use of production rule architectures has contributed far beyond the "logical" tradition, to nearly all areas of cognitive science including memory-based approaches. This point is elaborated in §1.4.2.i.

Strategies in Mathematics: Pólya's Process

György Pólya's *How to Solve It* (1945) stands as one of the most important contributions to the problem-solving literature in the twentieth century. Even now, as we move into the new millennium, the book continues to be a favorite among teachers and students for its instructive heuristics. (Michalewicz & Fogel, 2002)

The small book is a compendium of techniques that Pólya found useful, both personally, in his own career as an eminent mathematician, and pedagogically, through a prodigious amount of mentoring and collaboration. He starts by outlining a four-step problem solving process:

(1) Understand the problem

(2) Devise a plan

(3) Carry out the plan

(4) Looking back

"In order to solve this differential equation you look at it till a solution occurs to you," he jokes, introducing the first step. Understanding the problem is the same first step as in the Gestalt process, where "eyesight is insight" (§1.1.1). Pólya asks "How do the data relate to the unknown?", and he suggests introducing suitable notation or drawing a figure; these are all means for probing Gestaltist "structural features" of the problem.

Devising a plan, the next step, involves trying to find related problems that have been previously solved or known techniques or theorems that might be relevant (analogy, §1.1.4, and solution schemas, §1.1.3). When adapting and evaluating these sources to the current context — or if the solver cannot think of anything relevant — Pólya suggests a number of further strategies, including:

- **Does it use everything?** A good solution will often incorporate all supplied information, or if not, a clear case should be made why some piece is irrelevant. By looking at unused pieces, a new insight might be suggested. (This technique is indirectly used in structure mapping (Gentner, 1983), where mappings using more information are preferred.)
- **Work backwards.** Focus on how the "unknown" quantity could be reached; this can cue previously solved problems which are superficially different but share a common method of solution. (This may trigger "deep" structural features, of the sort which characterise expert sorting and solving as in §1.1.3. The technique is also implicit in goal-driven production and planning systems, *e.g.*, where production rules are matched against the desired end state as opposed to the starting state.)
- **Generalise.** If a problem appears difficult, it may be because it is an unusual, special case of an easier, more general problem.
- **Try a more specific problem.** Example instances of the problem may be easier to solve, and these might suggest a generic solution.
- **Find subproblems.** Formulating prospective intermediate lemmas might help understand the problem domain, and by moving in small steps, the final solution might be easier to reach. "Wishful thinking" is a related technique where one sees a change to the problem that one expects will make it easier; in practice, this can lead to a subproblem. (These techniques are akin to subgoals in AI planning, §2.1.3.)

What is noteworthy is that Pólya explicitly concentrates on techniques for retrieving prior knowledge that will be relevant, with most of step (2) devoted to these techniques and most of the book devoted to step (2). (In contrast, most schema theory accounts appeal to "associationism" and leave it at that, and production rule models tend to use built-in mechanisms for selecting among hand-coded, domain-specific rules.)

"Carrying out the plan" is the next stage in the process, done only after the plan or strategy has been formulated and considered. Every step in the plan should be checked carefully, by multiple formal means if possible, and also compared against simple cases and intuition. Obstructions or new insights might suggest changes to the plan; if so, one should repeat stage two of the process rather than rushing into the changed plan.

Once a plan has been completed, Pólya stresses, it is important to look back on one's work. This provides an additional validation that the result is correct, but more significantly it can increase one's understanding of the solution (*cf.* the self-explanation effect, §1.1.3, and compilation/proceduralisation, §1.2.3). Where possible, solvers are encouraged to relate the solution to concrete instances, checking units ("always!") and seeing whether some simple instantiation can be easily verified (*e.g.*, $x = 0$, or using right angles). Solvers might look for simpler solutions, symmetries used, or some characterisation of the sort of problems where the solution might apply (*cf.* feature abstraction, §1.1.3 and §1.1.4.iii).

Cognitive Strategies

Although *How to Solve It* is intended for mathematics problems, "mathematics" can be interpreted quite broadly and Pólya's process can be applied even more broadly. The *Productive-Thinking Program* (Covington *et al.*, 1966), consisting primarily of the same ideas as in Pólya's book, used detective stories to train students aged ten to twelve. In a similar vein, *Patterns of Problem Solving* (Rubinstein, 1975) served as a textbook for a university-level course in problem solving at UCLA, attracting over a thousand students yearly for over a decade (Mayer, 1983). Although no studies conclusively demonstrate the effectiveness of these large-scale projects, their popularity is a powerful testament, and better controlled experiments (Bloom & Broder, 1950; Schoenfeld, 1979) found that teaching such general heuristics can improve problem solving ability.

Application of strategies to dissimilar problems tends to be more difficult, and evidence that teaching strategies supports "far transfer" is scarce (Mayer, 1983). "Thinking outside the box", as suggested by the Gestaltists, is a shift in emphasis from Pólya's book to more general cognitive training programs[10]. The Productive-Thinking Program tries to facilitate creativity through instructions to "keep an open mind" and "think of many new ideas", including "unusual" and "unlikely" ones.

[10] The point was not overlooked by Pólya, and one strategy for "devising a plan" is simply to "be ingenious". He describes the importance of creativity and discovery to mathematics in greater detail in other — longer and more philosophical — writings, including the two-volume set *Mathematics and Plausible Reasoning* (1954) and *Mathematical Discovery* (1962).

Critical thinking extends the strategies in yet another direction, stressing the importance of distinguishing between non-judgmental "exploration" (Halpern, 2003; Grudin, 1990) and "black hat" (de Bono, 1985) logical analysis. Logic it identifies as a strategy useful for many types of problem solving. In self-monitoring, one reflects critically on personal tendencies, biases, and recent activity so that habit does not restrict the breadth of strategies and knowledge one can bring to bear on situations (Csikszentmihalyi, 1990). Of course it is widely recognised that none of these strategies are a quick fix for expertise; as we have seen, expertise requires intense domain-specific experience, with chess masters familiar with "50,000 chunks" (§1.1.2) and expert cooks with 50,000 "kitchen facts" (Norman, 1980). Studies of musicians (Hayes, 1981), chess (Simon, 1980) and other fields (*ibid.*) has led to the so-called "ten-year rule", that it takes at least this much time, in addition to luck and regardless of domain-general strategies, to become an expert at *anything*[11].

In memory-based theories, these strategies — including the use of logic and self-analysis — can be viewed as schemas, and their acquisition can be accounted for by abstraction across repeated experience or by direct instruction. These accounts are, however, undeniably vague with regards to these high-level strategies. In production systems, their use is more concretely specified, but even here, although they can be conceptualised as "executive productions" (Newell & Simon, 1972; *e.g.*, means-end analysis, §2), their implementation is usually outwith the production rules framework or else specialised, *ad hoc*, for a specific domain. The means for choosing appropriately between high-level strategies is something which has received very little attention at either the theoretical or practical level (Michalwicz & Fogel, 2002).

1.4. Critical Evaluation

Over-generality and over-specificity are two of a number of common criticisms aimed at these approaches. Before reviewing them let us examine what, precisely, they are trying to achieve. Each theory ostentiably tries to aid understanding of problem solving, but what does this mean? There are many legitimate and distinct ways of contributing to this goal:

- **Pedagogical**. Can the theory assist human problem solvers?
- **Modelling**. Can the theory predict human behaviour?
- **Physiological**. Can the theory explain how the physical structure of the brain facilitates problem solving?
- **Computational**. Can the theory be used to build artificial (*e.g.*, machine) systems?
- **Philosophical**. Can the theory offer vocabulary which assists in the other aims?

A "complete" theory of cognition will give affirmative answers to each of these questions — and, to be sure, many more which are not listed. It is clear from the limited survey in this chapter that the field is a long way from such a "complete" theory, but each experiment and every new approach sheds new light on at least one of these questions.

---

[11] The rule, of course, is facetious but the principle stands.

1.4.1. Levels of Description

A common philosophical distinction in the literature is the identification of three levels of organisation within cognitive systems:

- **The Semantic (or Knowledge) Level.** At this level we explain why people, or appropriately programmed computers, do certain things by saying what they know and what their goals are and by showing that these are connected in certain meaningful or even rational ways.
- **The Symbol Level.** The semantic content of knowledge and goals is assumed to be encoded by symbolic expressions. Such structured expressions have parts, each of which also encodes some semantic content. The codes and their structure, as well as the regularities by which they are manipulated, are another level of organization of the system.
- **The Physical (or Biological) Level.** For the entire system to run, it has to be realized in some physical form. The structure and the principles by which the physical object functions correspond to the physical or biological level. (Pylyshyn, in Posner, 1989)[12]

In computer science, the physical level is the hardware; the symbol level may be viewed as the assembler instructions and byte codes; and the semantic level as high-level programming languages (although often the semantic level is taken as extra-systemic). When looking at the various aims of cognitive theories, we can align pedagogical theories with the semantic level and the physiological with the physical. Theories with philosophical aims might apply at any of the three levels. Predictive and computational approaches involve all three, but their grounded inspectability places them primarily on the symbol level. Semantics, in the practise of these approaches, is usually hand-coded for a particular domain and not theoretically important, and the physical level is usually dictated by available machinery (in the psychology world, almost invariably a computer[13]). Specifying all three levels leads to a unified cognitive architecture, a framework for modelling a wide range of cognitive tasks. this philosophical approach will be discussed in §1.4.2. For now, let us note that this approach is promising — because it gives simultaneous answers to many of the questions set out in the last section — but also potentially confusing — because of the breadth of answers they provide (design choices which are merely incidental at one level, *e.g.*, the use of XML, might be taken as definitive answers at another, *e.g.*, the conclusion that concepts in the mind can be defined by a DTD).

The strategies in §1.3 and memory-based theories in §1.1 have tended to focus on pedagogical and philosophical questions. Of these, Gestalt theories address the semantic level almost exclusively and schema theories do so primarily, although some concrete demonstrations have addressed the symbol level (*e.g.*, SME, MAC/FAC, ACME; Forbus *et al.*, 1995; Hummel & Holyoak, 1997). It is production architectures which have given the most attention to the symbol level, yielding a compelling means of answering the predictive question without neglecting a thorough account at the semantic level. The resulting cognitive models will be reviewed in the next section.

---

12 These levels are closely related to Marr's "computational", "algorithmic", and "mechanical" levels (1982).

13 There is an enormous literature on neurophysiological models (see, *e.g.*, Churchland & Sejnowski, 1992) which is beyond our present scope. Such "bottom-up" approaches to cognition will hopefully one day meet the "top-down" psychological approaches covered in this chapter, but the gap between them is currently quite large. One promising development has been the AI technique of neural networks, introduced in §2.2.2.ii, which has influenced "spreading activation" models used in some systems, both analogical and production rules.

1.4.2. Cognitive Modelling

A principal objection to many of the accounts described so far is that while they may be appealing, they are frequently too vague, too general, and/or too convoluted. As abstract theories "explaining" cognition, they are "non-scientific" according to observations of Popper (1963):

> (1) It is easy to obtain confirmations, or verifications, for nearly every theory — if we look for confirmations.
>
> (2) Confirmations should count only if they are the result of risky predictions; that is to say, if, unenlightened by the theory in question, we should have expected an event which was incompatible with the theory — an event which would have refuted the theory.
>
> (4) A theory which is not refutable by any conceivable event is non-scientific. Irrefutability is not a virtue of a theory (as people often think) but a vice.
>
> (7) Some genuinely testable theories, when found to be false, are still upheld by their admirers — for example by introducing *ad hoc* some auxiliary assumption, or by reinterpreting the theory *ad hoc* in such a way that it escapes refutation. Such a procedure is always possible, but it rescues the theory from refutation only at the price of destroying, or at least lowering, its scientific status. (I later described such a rescuing operation as a "conventionalist twist" or a "conventionalist stratagem".)

Confirmations are held up for all the theories, but this comes far from proving them (insofar as "proof" is even possible outwith an axiomatised domain); *ex post facto* supporting observations are empirically invalid, and as far as experiments to "test" theories, surprisingly few are "risky" in the sense of (2). The strategies of §1.3 are undoubtedly a useful summary of techniques, but they do not comprise a scientific theory — Pólya and other proponents never attempted to claim that they did. Schema "theory" introduces vocabulary that many have found attractive, but for its vagueness, we feel, it must be struck down by (4). Production systems, having undergone such significant "evolutions" and "revisions" in response to numerous "refutations", can likewise be attacked on the grounds of (7). If "the scientific status of a theory is its falsifiability, or refutability, or testability" (*ibid.*), how are we to evaluate any of the approaches in this chapter on any of the grounds introduced in §1.4?

A second objection is ambiguity between "strong" and "weak" equivalence. A theory could be realised in a testable model and demonstrated in a "risky" experiment, but obtaining results that correspond to human performance still does not imply that the underlying mechanisms are the same. In "weak equivalence", a model is acceptable if it provides the same "output" as human subjects for a given "input" set. Assuming expert (or even idealised) human performance, this approach characterises many of the AI systems covered in the next chapter (and understandably so, for these systems are judged almost exclusively on how well they solve problems). The method by which the results are reached in the model might bear no resemblance to human cognition; it might even be a "black box" whose method even the developers do not understand. "A stronger claim might be that the model realizes some particular function using the same *method* as the person being modeled" (Pylyshyn, in Posner, 2005). For any theory purporting to explain human problem solving in an understandable fashion, this *method* must be explicit and this "strong equivalence" is essential. Unfortunately, it can be very hard to establish.

The Theory / Implementation Divide

Researchers can — and have begun to — overcome these objections by constructing "cognitive models" which offer both a theory and an implementation, with a clearly distinguished line drawn between them. The theory, "above the line", should set out the conceptual explanation for the phenomenon is being addressed — we will take problem solving as the example. The theory should be concrete enough to be implemented, tested, and falsifiable; but beyond this it should retain as much generality as the authors envisage.

Illustrations provide some insight into what is meant by a general theory and have been common in the psychological literature since Aristotle. They are typically confirmations in the sense of Popper's criticism (1), rather than a systematic attempt at scientific verification. By providing a complete, fully-specified implementation, "below the line", a model allows the empirical exploration of the theory. To be scientifically useful, this exploration must include the sort of risky experiments in (2) and leave itself open to the prospect of failure in (4). Moreover, constructing models (and making them publicly available) facilitates the sociological factors that determine the scientific *Zeitgeist* (Kuhn, 1962), including independent verification, replication in other domains, and ultimately widespread adoption.

Unfortunately, implementing a theory is rarely straightforward. Computers are currently a long way from realising the parallel processing of billions of neurons (and a model so complicated would likely be an uninspectable black box anyway), so it is frequently necessary to approximate aspects of a theory when working below the line. "Simplifying assumptions", as they are often called, are a key part of a cognitive model, allowing many different implementations without impacting what authors propose to be a theory's full generality. If the model fails in experimentation, an author can, of course, stand by the theory and blame the simplifying assumptions; but in the philosophy of cognitive modelling, a theory with a flawed implementation is no worse than a theory with no implementation. Both are non-scientifically vague, and the former, at least, admits it.

As an example of the analysis that the cognitive modelling methodology affords, let us look again at the General Problem Solver (Newell & Simon, 1972). While Newell & Simon propose production rules as a very formal technique, their notion of "strategies" is a very general one. Part of the theory leads to an extraordinarily powerful system, and the implementational details — such as the use of the IPL programming language — do not preclude other varied implementations. Other parts of the original theory (*e.g.*, change representation, choice of strategy or heuristics for control) were vague and either unrepresented or hand-selected in the implementation. These aspects of the theory, the authors make clear, were not the focus of the project[14]. Recent systems, ACT-R and Soar, have taken the original theory "above the line", extended it, and

[14] We propose extending the line metaphor, twisting it a bit, to say that unrepresented parts of a theory are "to the left of the line" and hand-coded parts of a theory are "to the right of the line".

provided alternative, much more complete implementations. In so doing, they have founded the approach of "cognitive architectures", models where the theory and the implementation are together available as frameworks within which other cognitive models can be developed.

By requiring the implementation of some version of a theory, the discipline forces at least one concrete interpretation. As cognitive theories are frequently very general and terms overloaded, this helps communicate what is intended. At times, it can also help identify tautological claims (*e.g.*, claiming "schemas are the fundamental unit of thought", after defining them as "the fundamental unit of thought") and facilitate meaningful hypotheses ("cognitive architectures can be realised by treating experience, abstractions, and concepts in a uniform way"). Commonly, the creation of a model highlights weaknesses in a theory, forcing the author either to remedy them or to acknowledge them.

Psychology straddles between science and philosophy, and many unscientific "theories" have had great import despite vagueness and non-falsifiability. The chief value of cognitive modelling, in our view, has been to bolster general philosophical approaches with scientifically testable models, without limiting the generality of the theory. Such models can support — or challenge — weak equivalence hypotheses:

- Models find the same solutions as the people they are modelling.

They can also provide evidence for strong equivalence in many ways:

- Models reproduce errors typical of humans.
- Slight variances in a model correspond to individual differences in people's performance in a task (a generalisation of accounting for errors).
- Models match timing data for how long people take on different problems.
- Models can be inspected to show intermediate steps in the process which correlate with human problem solving activity (*e.g.*, eye-tracking and speak-aloud protocols).

Cognitive models have made use of each of these types of evidence, as well as many others. ACT-R models have an especially broad base of such support, providing accounts ranging from mistakes in arithmetic and saccades in word recognition to capacity and activation in associative memory. Yet however extensive this evidence is, it can only be described as circumstantial; it can never prove that a model is strongly equivalent. The introduction of simplifying assumptions, essential as it may be, further separates theories from their models, and means that sociological factors will be the overriding determinant of acceptance[15]. These sociological factors, including perceived utility and appeals to elegance, can be greatly enhanced through cognitive models, of course, and it is unlikely that the psychology of problem solving would have advanced as much as it has without the specificity demanded by cognitive modelling. By drawing a line between general theories, with philosophical appeal, and specified implementations, with predictive and computational utility, cognitive modelling broadens the bases of evaluation: the line serves, orthogonally, as a bridge between theory and implementation.

---

[15] This conclusion has been applied to the entirety of science (Feyerabend, 1987) but we feel it is particularly strong as regards the psychology of problem solving.

The Turing Machine Critique

Despite their concrete implementation, cognitive models are not immune to being over-general. Any model satisfying a small set of computational properties is isomorphic to a Universal Turing Machine, and is therefore capable of any finite state computation (Turing, 1936), subject only to limitations of memory and time. Models for specific tasks usually do not have these properties, but most cognitive architectures, including GPS, ACT-R, and Soar, do have the necessary properties. It would be difficult to imagine any model of schemas which did not also fall into this category.

This does not invalidate many of the benefits of cognitive modelling: specificity and understanding as well as a greatly increased base for evaluation. What the Turing Machine critique emphasises, however, is that final arguments for theories or models can never be made on their performance — in weak or strong equivalence — alone. The case must also be made that the model transfers, or the architecture applies, with minimal modifications, to alternative domains. For most models in the literature, this case has not been made; even with ACT-R, perhaps the most widely used such framework, there is a great deal of domain-specific programming involved to model any novel problem. In our own experience, the use of production rules can lend itself to persuasive timing results (albeit normally not without some tuning), but the cost in development effort is extensive, particularly as compared with other Turing Complete languages such as Java or Lisp.

Implementing the strategies described in §1.3 presents additional challenges in nearly all cognitive modelling environments. Although Newell & Simon reiterate the use of such strategies (1972), the implementation of the strategies has nearly always been done as an integral part of the system — and not in the terms of the productions the system advocates. This has been the case from early means-end analysis (*ibid.*) through to the recent analogy learning mechanism (Anderson & Lebiere, 1998). In the few instances where a general strategy has been implemented as an explicit cognitive model (Salvucci & Anderson, in *ibid.*; Murdock *et al.*, 1998), the projects required large sets of productions for retrieval (restricted in all the models to exact matches of a very small number of explicitly named entities) and structural mapping (which again looked at only a limited number of named entities in fixed or nearly-fixed positions). Prospects for their general applicability are daunting:

> In a scaled up version of this model ... this mechanism would become overwhelmingly cumbersome and would not generally support tractable retrieval. (Murdock *et al.*, 1998)

Performing Sternberg's (1977) basic "`A:B::C:D`" analogy task (over four binary attributes) in ACT-R required over 1000 lines and 25kb of *ad hoc* source code; what is more, this was based on a pre-existing model and held up as a model example to "illustrate how the [pre-existing] model can generalize to other tasks" (Salvucci & Anderson, in Anderson & Lebiere, 1998). Each of these efforts are unquestionably impressive feats; however, there has been no attempt to re-use

any of them in other problem solving contexts, and without applications to multiple domains it is difficult to make the case that they are anything other than computer programs in an obscure language, and it is impossible to make the case that any of these are general models of analogy. Where cognitive architectures are Turing Complete, modular re-use is an important sociological factor in their evaluation.

To realise the stated goal of "a unified theory of cognition" (*ibid.*), it must be possible to maintain a vast number of models simultaneously, with appropriate knowledge retrieved and re-used efficiently. Under certain circumstances, it must be possible for this prior knowledge to be applied spontaneously in novel domains ("insight" or "far transfer", §1.1). From a Universal Turing Machine one should expect nothing less! ACT-R appears to us to come closest to the goal (although we strongly dispute proponents' claims (*ibid.*) that it has been achieved), yet it suffers from intractable scalability problems (Murdock *et al.*, 1998) and its capacities for far transfer (Müller, 1999) and abstraction (Anderson & Schunn, 2000; Eysenck, & Keane, 2004; *v.a.* §ActRContextualPriming) are far from adequate. In nearly all current cognitive frameworks, the composition and transfer of individual models is exceedingly difficult, but until this becomes both easy and widespread, the methodology of cognitive modelling is incomplete.

The Critique of Curious Domains

A further difficulty in the use of cognitive models is that frequently the simplifying assumptions narrow the scope of application. Focussing on certain problem domains is necessary, but the validity of the theory is bolstered only to the extent that experiments are risky (Popper's criticism (2)) and, especially for Turing Complete implementations, to the extent that the selected domains are representative of a larger class to which the theory and model generalise. In many instances, however, the simplifying assumptions tacitly restrict the classes of problem domains where the model is applicable:

> The researchers [from the Gestalt School through to many current practitioners] made the underlying assumption, of course, that simple tasks such as the Tower of Hanoi captured the main properties of "real world" problems, and that the cognitive processes underlying participants' attempts to solve simple problems were representative of the processes engaged in when solving "real world" problems. Thus researchers used simple problems for reasons of convenience, and thought generalizations to more complex problems would become possible. Perhaps the best-known and most impressive example of this line of research remains the work by Newell and Simon (1972). (Wikipedia, "Problem_solving")

What has since become clear is that these generalisations are not as easy as many researchers thought. The components necessary for human expertise (§1.3.ii) would also be necessary in any corresponding "strong" cognitive model — *viz.*, 50,000 domain-specific facts and ten years' experience (or equivalent). In a hypothetical such model, is the "above the line" / "below the line" distinction relevant? In practical terms, would any result justify the work involved in coding such a model?

The answer may lie in yet another of Pólya's strategies (1945) — "lazy thinking" — or the aphorism — necessity is the mother of invention[16]. The only feasible solution is the "lazy" one of letting the cognitive model acquire the facts from experience. Being lazy, however, is not easy. By focussing mainly on simple tasks, problem solving architectures have largely bypassed the problem of large-scale learning from experience. The cognitive architecture with the most extensive capabilities on this front appears to be ACT-R, and even they are severely limited, as mentioned in the last section. Furthermore, the theoretical aspects of learning in ACT-R, "above the line", are not clearly distinguished from the implementation, and the implementation "below the line" is complicated to the point of near-opacity. It is doubtful whether ACT-R's learning mechanism would be suitable for the task and unlikely that any other cognitive architecture would fare any better.

1.4.3. Subjective Conclusions

Cognitive Modelling

In view of the critiques of the Turing Machine and curious domains, we suggest that one promising direction of research would be an alternative "below the line" implementation of a cognitive architecture. The project could adhere to a well-known current theory — such as ACT-R — but seek to address issues of abstraction, retrieval, scalability, and model re-use. In the philosophy of cognitive modelling, having multiple implementations of a theory serves two purposes: it expands the scope of the theory's demonstrated applicability, and it clarifies how much of an existing model is actually "above the line".

Another promising direction of research, it seems to us, is the construction of a cognitive architecture based on schema theory. These theories have been almost universally "non-scientifically" vague; in the relatively few attempts to flesh out memory-based theories with cognitive models (*e.g.*, MAC/FAC, ACME; §1.4.1), the focus has been on quite specialised theories and very domain-specific implementations. A general architecture, corresponding to ACT-R or Soar but implementing a schema theory view of cognition, is lacking and, in our view, is one of the chief causes of the under-specificity in alternatives to production architectures. Where models have been developed for memory-based theories, the results have been impressive (*ibid.*); an architecture could seek to extend this work. It would also, certainly, do well to take ideas from ACT-R and Soar, particularly about modelling associative memory.

How different would such an architecture be, we wonder, from ACT-R and Soar, or from an alternative implementation of their underlying theories? These systems have changed enormously since production architectures were first proposed: they have incorporated many memory-based ideas, and they have evolved to account for a vast range of experiments in perception, categorisation, and analogical reasoning. Our suspicion is that the use of "production rules" in any of these systems might well be found now to be vestigial — products of a pre-Object-Oriented programming world which can comfortably slide below the line or, in a modern re-implementation of the core theory, abandon their logical insistence of "`if ... then`" conditionality. Such a re-implementation would begin to look like a cognitive architecture based on features and schemas.

[16] *cf.* "If necessity is the mother of invention, then analogy is the father." (Ruth & Hannon, 1999)

Approaches to Problem Solving

Memory-based theories, in our opinion, have tended to be too vague to be useful from an applied, predictive, or modelling standpoint. Production rules have proven extraordinarily attractive because of their success in certain concrete domains with respect to each of these criteria. We concur with many authors, however, in being unconvinced that this success will scale, and we observe that memory-based theories have made notable pedagogical and philosophical contributions to problem solving, despite their vagueness.

It seems to us that the role of context may be far more pervasive than most theories acknowledge. Among the experiments in this chapter, wherever context was investigated as a factor, it was found to have an effect. The Wason Selection Task (§1.2.1), in particular, has been performed in an impressive variety of studies; one consistent finding has been that the use of thematic content ("If a person is under 21, he or she is not drinking alcohol.") leads to marked improvements in performance, particularly where the content is familiar. This has been explained by various theories (*e.g.*, "pragmatic reasoning schemas", Cheng & Holyoak, 1985; "task-semantic" interpretation, Stenning & van Lambalgen, 2001), nearly all of which align with the memory-based tradition.

Although these explanations could be modelled in production systems, reliance on numerous "`if ... then`" rules offers us very little (apart from laborious representations). Attempting to get by with a small set of such logical rules (§1.2.1, *e.g.*, Braine, 1978; Rips, 1994), on the other hand, seems wholly inconsistent with "real world" cognition:

> If one tries to describe processes of genuine thinking in terms of formal traditional logic, the result is often unsatisfactory: one has, then, a series of correct operations, but the sense of the process and what was vital, forceful, creative in it seems somehow to have evaporated ... there is the danger of being empty and senseless, though exact.
> (Wertheimer, 1945)

Soar and ACT-R, as described in §1.2.3, have explicit memory stores as core components. They incorporate prior knowledge through well- specified associative techniques (pattern matching and semantic nets, respectively); in practice, this may well be one of their main benefits (*e.g.*, Murdock *et al.*, 1998). The need for further separate internal techniques — heuristic mechanisms for conflict resolution and learning, not expressed as productions — strikes us as an inelegant (and possibly redundant) bit of shoehorning.

Prior knowledge is the fundamental ingredient in human cognition. From this ingredient, the formation of concepts — meaning — is the fundamental action of cognition. The simplest coherent theory of concept formation, in our view, is that of explanation-based categorisation (§1.1.1.i; Murphy & Medin, 1985). A schema ("a data structure for representing the generic concepts stored in memory", §1.1.2; Rumelhart, 1980) captures this notion in its generality, and induction by analogy suggests a compelling means for how they may be formed (Gick & Holyoak, 1983; Gentner, 1983). In problem solving, the recognition of certain features, both as superficial attributes and structural relations, seems to associatively cue retrieval of schemas and experience

which govern human behaviour (Chi *et al.*, 1981; Ratterman & Gentner, 1983). Based on this outline, we would expect fundamental entities in a cognitive architecture to be:

Schemas: experiences, concepts, and intermediate abstractions

Features: the recognition of schema-based concepts in a particular problem instance

Instead, when we look at existing architectures, they are nowhere to be seen. This is surely a mistake!

We note with wonder the historical primality of Associative theories of mind: after two millennia, the theory and terminology are recognisable and relevant. We observe that the definition of "schemas" accommodates concepts, templates, and high-level strategies for problem solving. Although some argue that its definition is so general as to be unhelpful, we believe that it is usable (as demonstrated in §3.2), and we further believe that past attempts to enforce further high-level distinctions have been more misleading than helpful (*e.g.*, abstract "solution schemas" versus "worked examples", §1.1.4; or ACT-R's unusual interpretation of the declarative/procedural split, §1.2.3). A simple unifying theory that still needs specification is preferable to an over-encumbered theory which has been complicated numerous times in response to successive objections and flaws. One immediate benefit of this generality is that such memory-based schemas can represent logical rules and productions, but the converse is not true. Production approaches uniformly rely on complex control strategies that are not implemented within the framework of the theory. Associative secondary laws — *i.e.*, similarity (through the presence of identical features) and availability (vividness, frequency, and recency) — offer a theoretical underpinning for schema selection which seems much stronger than the heuristics (and innumerable parameters) which have evolved for the selection of production rules. Production rule architectures, furthermore, rely on external mechanisms for learning, make representing concepts difficult (especially theory- or explanation-based categories), and are clumsy or incomplete when explaining the effects of context; yet somehow they are extraordinarily popular. We can only attribute this to their focus on fully specified, usable cognitive models: sadly, there is no implemented alternative.

Although memory-based theories are often criticised for being underspecified, we believe that they are now mature enough to encourage an attempt at a "below the line" cognitive architecture. The use of features in cueing schemas seems an area that is particularly well described and powerful enough to merit the development and exploration of a problem solving framework. Efforts to do so, we are sure, would both further understanding in the area of problem solving and focus research questions on aspects of cognition that remain underspecified.

Chapter 2. Computational Approaches to Problem Solving

In contrast to psychological theories of problem solving, where goals are nearly as diverse as the ideas, computational approaches have had a clear and narrow focus: results, results, results. In this chapter, we will give an overview of many of the principal techniques used in problem solving with computers. The field is vast, and we will necessarily omit some techniques, but we will endeavour to cover the most pervasive and influential ideas and give references where we have had to make omissions. We also cover in extra detail those approaches which, although not as common as others, are specifically related to the memory-based paradigms introduced in §1.1. As a means of tying many of the computational approaches together, including production rules and planning, and to give background on one problem solving domain relevant to our experiments, we review systems for theorem proving in §2.3.

One significant topic that we will cover only in passing, despite its great importance, is how knowledge can be represented in a computational system. We will typically assume the use of predicate representations (*q.v.*, §1.ii) with fixed domain-specific meanings, as is common in many systems, but this is in no way intended to limit the immense spectrum of techniques that have been used for representation and semantics, nor the potential applicability of other representations for the techniques described here. (For comprehensive coverage of issues in knowledge representation, the reader is referred to Brachman & Levesque, 2005.)

For some types of problems, algorithms can be completely specified which will solve them reliably, and for many more problems, established mathematical techniques (*e.g.*, Apt, 2003) can be used off-the-shelf to deliver automated solutions. Where problems are easily solved through such algorithms or as domain-specific programs, we judge that it is the human developer who has actually "solved" the problem, in a generalised (algorithmic) fashion. While these solutions are powerful and worthy, they are relevant to our present purpose only where they suggest techniques that apply to a varied set of problems, leading to computational solutions far beyond what the initial developers may have considered. Consequently this review looks only at generally applicable techniques and not at the wealth of libraries implementing these techniques (and others) for particular programming languages. (The reader is referred to Michalewicz & Fogel, 2002, for a review of implementations and for a good account of certain canonically "hard" problems — *e.g.*, travelling salesman problem.)

2.1. Logical Techniques

2.1.1. Production Systems

The discipline of Artificial Intelligence (AI) has been profoundly shaped by the work of Newell & Simon (1972) presented in §1.2.2 as an exploration of *Human Problem Solving*. Production rules — irrespective of their cognitive applicability (§1.4.3) — have an "`if ... then`" structure which fits neatly with the logical architecture of CPU's. This fact, together with the compelling capabilities of early production rule systems, has led to an explosion in their use in computational problem solving. Current definitions of production systems typically identify three main components:

> **The Set of Production Rules**: the basic knowledge unit is the "production", *i.e.*, "a [pre]condition-action pair [which] defines a single chunk of problem-solving knowledge" (Luger, 2005)
>
> **Internal Representation**: "the current state of the world" or "problem state" is described symbolically; also called "working memory" (*ibid.*)
>
> **Control Plane**: the selection ("firing") of productions and corresponding changes to the problem representation are managed by a system-specific process

The "action" part of the production rule updates "working memory": in the simplest instance, "operator productions", defined by the problem domain, describe the valid elementary changes in a problem state. More complex production rules might express the result of several operators applied sequentially, or in some systems, enrich the problem state but without fundamentally changing it, *e.g.*, by updating a "blackboard" with notes which might enable the conditions of other productions. Predicate logic (§1.ii) is often used for the problem representation and for specifying preconditions, and in "rewrite rules" which specify how the action changes the problem representation.

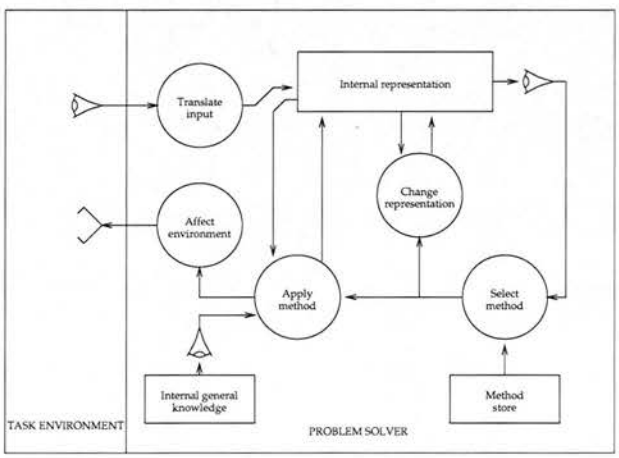

FIGURE 2.1: **General Problem Solver Organisation**. GPS is one of the earliest and most influential production rule systems. In this model, the solver iteratively applies methods to the problem until an answer is found. The rectangular entities correspond approximately to the three components described in the text. (adapted from Newell & Simon 1972)

The control plane is the most varied component, as was observed in cognitive architectures (§1.2.3). Commonly, it follows a "recognise-act cycle", where a production is chosen and applied to the problem state, with the process repeating on the new problem state. A "conflict resolution" strategy is usually provided to choose uniquely if multiple production rules have validated conditions, and some systems also provide a pattern-matching strategy (*e.g.*, the Rete discrimination network algorithm, Forgy, 1982) to optimise the process of checking conditions. Beyond this general description, however, there are numerous mechanisms used for control in AI production systems, including general techniques for search (§2.1.2) and planning (§2.1.3) as well as — for all but the simplest problems — domain-specific heuristics (§2.1.4).

Finite States

One attribute common to many production systems — particularly early ones — is the reliance on "well-defined" (deterministic) problems, where a problem is defined by an explicit initial problem state, a goal state, and a finite set of operators which change one problem state to another. Tower of Hanoi and noughts-and-crosses are two well-studied examples. An attraction of this type of formulation is that the resulting state space can be easily defined mathematically: it can be represented as a directed graph where the origin corresponds to the initial state, edges correspond to operators, and a vertex at the target end of an edge is the state that results from applying that operator. The space of all reachable problem states is then the set of vertices with which the origin is path-connected. If the goal state is in this set, any path leading to it corresponds to a solution, and the shortest such path is the optimal solution.

For many problems the set of problem states is finite, and even when it is unbounded, formal guarantees of completeness — that a solving technique will find a solution, if it exists — can be provided. Within finite state domains, explicit bounds can be given for time and memory use, and any solution is assured to be found within these bounds. Conversely, if a problem cannot be solved, the system will be able to demonstrate this (in theory) by exhaustively enumerating the state space. Such properties can be extremely useful in computational systems.

Expert Systems

As discussed in the critique of a curious domain (§1.4.2.iii), not all problems possess the nice structure characterized by a finite state space, and many which do have such large theoretical bounds that they are of no practical use. Problems in the real world, it has been found, frequently have this "ill-defined" or inordinately large character: any feasible solution technique, in many instances, will require the use of a great deal of domain-specific knowledge. Production rules have been adopted for such domains by combining a great many facts — each expressed as an "`if ... then`" rule — and scalable precondition pattern-matching techniques. This comprises an "expert system".

Dendral, one of the earliest such knowledge-rich systems, took hundreds of rules observed from how stereochemists analyse spectroscopy data to solve problems in molecular structure. Mycin, in the field of medical diagnostics, has expressed thousands of concise medical facts as productions to infer causes — and treatments — relating to blood disease. (Wikipedia) From these progenitors, two frameworks are especially popular in current usage. Prolog (Clocksin & Mellish, 1994) is a programming language closely related to the precondition-action distinction, working backwards and adding support for generality in rules through unification-based instantiation; it is often considered one of the core theoretical systems in AI and is used in numerous knowledge-rich applications. Finally, CLIPS, "the C language integrated production system", offers object-oriented support for production rules and selection and customisation of conflict resolution strategies, and is "probably the most widely used expert systems tool" currently available (Wikipedia).

Despite their frequent successful performance, expert systems have faced and continue to face serious challenges. Mycin was able to outperform most physicians, but the medical community was not prepared to embrace it: even though it could present the causal chain of rules leading it to its conclusion, the chance of an errors therein could not be eliminated. A mis-diagnosis by a "thinking machine", even if it is statistically less likely than a mis-diagnosis by a physician, is for many, a far more disturbing prospect. Thus human review was and is still necessary, in any mission-critical setting.

These prejudices against machines are not entirely unfounded; because systems are only as good as the rules they are given, for any infinite real-world domain, they will be necessarily incomplete. Human experts can compensate through techniques such as "far transfer" (§1.1.4) and structural understanding, but expert systems flounder. Mycin, for example, has been known to ask patients whether they are pregnant, even if a patient has already indicated that he is male. Such "ignorance" would not endear the system to patients, and although that individual issue could be repaired, there is no such thing as "total knowledge" in a field such as medicine. Expertise requires, in addition to a great many facts, good common sense, specifically conceptual understanding and strategies for problem solving; however "expert systems [apply] heuristics in inappropriate situations, such as when a deeper understanding of the problem would have indicated a different course" (Luger, 2005).

2.1.2. Search

"If at first you don't succeed, try and try again." One of the most basic strategies in solving problems, if the answer is not immediately apparent (such as a unique production rule match) is to try different possibilities. The processing power of the computer magnifies the reach of this approach, and the atomic nature of production rules means it is not difficult to implement.

Newell & Simon (1972) popularised the technique of searching a state space to find a path to the goal state. This can be done depth-first — applying productions sequentially until none apply, and then backtracking to the first alternative (common in Prolog) — or breadth-first, where all productions are tried against the initial state, then against all adjacent states, and so forth (guaranteeing the shortest path to a solution).

For all but the simplest domains, however, these approaches both lead to combinatorial explosion, as the number of states can grow exponentially with depth. To overcome this problem, search can be "informed" by heuristics which encourage selecting likely productions and, in best-first search, focus on the problem states which appear closest to the goal. Strategies such as "hill-climbing" or "means-end analysis" estimate this proximity to the goal and are demonstrably effective for a wide range of problems (*ibid.*). They can be used in conjunction with a wide range of different search techniques as well (Pearl, 1984), specialised for various problem domains.

The drawback of these heuristics, however, is that they are inherently representation- or domain-specific. A perfect heuristic is tantamount to solving the problem, and even finding a useful heuristic entails sophisticated knowledge of the domain: both are nearly always the province of the human developers.

2.1.3. Planning

When interacting with a non-deterministic real world, a more flexible approach to problem solving is needed than search alone. The area in AI of "planning" has historically concentrated on independent machines — "intelligent agents" such as robots — which must operate in environments with uncertainty. The actions they choose affect this environment, and the agents themselves must be adaptable to unexpected encounters. This is facilitated by making actions part of a larger plan, formulated so that it is robust enough to cope with probable intermediate outcomes and reparable in the face of many other events.

Whilst search is an important component of planning, the need to be easily adaptable has introduced important strategies into AI problem solving. A major component in most planners is the identification of intermediate subgoals which lead to the final goal. By doing this, an agent can often respond better if the plan breaks down: if a new plan can be found leading to any of the subgoals, the remainder of the original plan can be reused.

STRIPS (Fikes & Nilsson, 1971), the Stanford Research Institute Problem Solver, is one of the first attempts at such an "embodied" problem solver, using a stripped-down predicate logic to

represent "the world" and the operators (precondition-postcondition rules) which can be used in the world. The STRIPS language has been used for a wide variety of planning problems, although modern planners such as Prodigy (Veloso *et al.*, 1995) and SHOP (Nau *et al.*, 1999) use more complex description and temporal logics or propositional logic with SAT-solving techniques.

In specific problem domains where planners are used, control of the process uses many of the same techniques as systems previously described; some of these will be covered in the next section. In the context of domain-independent planners, some important general problem solving techniques have been developed. The first of these, partial-order planning recognises that "goals" can sometimes be decomposed into mutually-independent sub-parts; these subgoals are usually simpler to solve than the original, and when effecting the plan, recognising that the subgoals can be resolved in either order — or even concurrently — leads to greater flexibility. The Hybrid STAN planner (Fox & Long, 2001), for example, can identify when specialised techniques might apply to individual problems; by having decomposed a complex goal, these amenable problems are more easily recognised.

A second general technique which has emerged is the use of hierarchical task networks (Nau *et al.*, 1998): instead of solving the problem by searching in the space of atomic actions, this approach analyses compounds of actions (usually performed sequentially) and dependencies between them. This facilitates planning at a larger granularity, bottom-up, and also the hypothesising of likely intermediate steps, top-down. This domain-independent technique has proven useful in many real-world domains and has resulted in, *inter alia*, one of the leading Bridge programs (*ibid.*).

2.1.4. Guidance and Control Strategies

The problem solving approaches described thus far in this section all use the general framework of specifying the operators available in a domain and, for a given problem, trying to find a sequence of operators to lead from an initial state to a goal state. We have thus far described several strategies for achieving this using search: working forwards from the initial state, backwards from the goal state, or fixing one or more intermediate states which break the problem into sub-parts, hopefully more easily solved. However, the choice of which of these to do — and, crucially, how to do it — is typically very difficult in practice. With fast computers and small state spaces it may not matter, but for most complex problems this choice becomes the central question facing computational solving systems.

A common solution is to allow the programmer to represent fixed orders in which productions should be tried. In Prolog, for example, productions are tried in the order they were entered. Other systems may use other strategies encoded in the system (*e.g.*, the Generalised Problem Solver), or encoded by the user as an extension to the system. The approaches vary with programming languages, production systems, and planners, but a very common one essentially uses expressions in a temporal logic or a regular expression grammar, *e.g.*, using "tacticals" in

theorem proving (Gordon *et al.*, 1989). As an example of the regular expression syntax, we might write "$A(B+)((C|D)*)$", meaning apply A once, then apply B as many times as possible and at least once, and lastly try C, or if that fails D, as many times as possible. An attraction of this approach is that where they exist, exact solutions (such as decision procedures) can often be concisely expressed in this way. This approach can also be used to express plans, *i.e.*, standard ways in which problems can be decomposed, by permitting holes or "gaps" where the strategy does not specify a technique for several steps, although it may be resumed later.

Another common means of controlling the selection of productions or search is to rank candidates according to some weight function. This might be an internal part of the system, preferring productions which have a history of success or which have been used recently (as in ACT-R, §1.2.3); or this might be a developer-supplied heuristic which evaluates the predicted utility of applying a production. This domain-specific "evaluation function" is similar to heuristics used in best-first search (*q.v.* §2.1.2), but invoked earlier in the process and used to inform the selection of productions or operators. (It is possible — and not uncommon — to use the same heuristic for both ranking productions and guiding search.) The domain-independent heuristics, as noted in §1.2.3, are reminiscent of the Associative secondary laws (§1.i); the domain-specific heuristics, likewise, can be viewed as a programmed association between the evaluated problem components and the relevant productions.

Unfortunately, these control strategies are nearly always implemented in *ad hoc* code; they are usually written at the level of the system as an extension to the system, rather than within the system as a module of knowledge for a given domain. Control knowledge — knowing when to use certain rules or operators — is an important part of domain knowledge. For problem solving, it is arguably the most important part of a system, but there is little standardisation in how it is represented. Procedures in programming languages can be very effective for individual problems, but they can be very hard to read; these strategies are typically kept separate from the core domain knowledge (*e.g.*, production rules), and are much more difficult to understand and re-use. We will return to this issue in §2.4.

2.1.5. Learning

One important aspect of a problem solving system is the prospect for it to improve its performance on the basis of experience or training. One technique has already been mentioned: by adjusting the likelihood of a production firing in response to past performance, a system can learn which productions are most effective in certain domains. Other heuristics — *e.g.*, evaluation functions — can also sometimes be tuned in this way, depending on their design: a large number of statistical and evolutionary methods have been used for this purpose. (These learning techniques are used in many other contexts, as well; some of these techniques stray from the logic-based focus of this section and will be reviewed again in §2.2.2.ii and §2.2.3.)

Chunking

Another learning technique common in production and planning systems is the "chunking" of atomic steps, introduced in §1.2.3. Where a system can identify a pattern of rules being applied in the same sequence, several times, it can improve performance by introducing a single "macro-operator" to perform the sequence. This technique is used in both ACT-R and SOAR (§1.2.3), and to a more specialised and powerful extent, in hierarchical task networks (§2.1.3).

A common variant on chunking is to do the sequence analysis, but then, rather than form chunks, store an n-gram probability model, recording the frequency of sequences of productions in a training set. With this n-gram model, the system can use history to inform the selection of productions: the previous $n-1$ steps give a probability estimate for selecting what should be tried next. This technique — and more powerful extensions such as Hidden Markov Models — are common in high-performance production systems as well as many bespoke systems for specific problems (*e.g.*, speech recognition).

Induction

A more powerful — but more difficult — learning technique is to attempt to synthesize compound tactics, *e.g.*, "$A(B+)((C \mid D)*)$", from a corpus of solutions or from experience. In contrast to chunks or probabilistic models, these tactics use tacticals which enforce a logical structure on the resulting plans or control strategy. Ideally, they will isolate a small number of sophisticated tactical expressions that generalise to a large class of problems; by finding these tactics off-line, systems can potentially skip the expensive step of searching through all atomic productions. This approach has been used in game-playing (Heneveld *et al.*, 1999) and theorem proving (Duncan, 2006), as well as many other areas. It can also be learned to generalise plans from solution sets, *e.g.*, a compound tactic "$A(.*)((C \mid D)*)$", where the "." stands for an unspecified step (either an operator production or another tactic).

In practise, however, this type of analysis is typically very time-consuming, and the results are not always useful. The complexity of looking for inductive generalisations in a set of N solutions is $O(2^N)$, both in time and in memory usage, as opposed to $O(N)$ time and $O(N^k)$ memory for chunking models (of length k). Furthermore, the results of such analysis can include a lot of noise. While "$(C \mid D)$" above might represent a useful heuristic, it might instead represent two very different and unrelated solutions. Without taking contextual factors into account, and specifically without a "deep" understanding of the underlying causal structure, the generalisations will be of limited assistance. Moreover, if solution steps are independent — for example, if "$A(B*)$" and "$(C \mid D)$*" act on separable parts of a problem — the tactical expression combining these two words is irrelevant. Most systems would function better if they are kept separate, but without very sophisticated domain-specific analysis, it is very difficult to prevent these compounds, and the time- and space- complexity is potentially factorial. Finally, the technique offers little insight into the fundamental question of when a strategy or tactic should be used. An analysis of the underlying problem states is necessary to have meaningful induction, and even though other techniques can help — notably evaluation functions — this context analysis then becomes an integral part of computational problem solving, which production and planning systems do not specifically address.

2.2. Non-Logical Techniques

The approaches described so far, modern incantations of logic-like rules and finite state search, go a long way towards solving a lot of problems, but where they fail, they do so in a major way. Many other approaches have been devised, and although none of them currently have the breadth of application of search and production systems, they do offer promise where search and productions do not, particularly when the approaches are combined. Unsurprisingly, many of these other approaches mirror some of the non-logic-based theories described in Chapter 1 for analogy and classification. Formal specification for these approaches is necessary for implementation in contemporary software or hardware, but we distinguish these approaches from those in §2.1 in that the use of logic is incidental: one might imagine these techniques running just as well on computational hardware not based on boolean bits — such as the brain.

Specialised logics can be developed to describe some of these approaches. In fact, this has been done, such as fuzzy logics where values are real numbers, and Bayesian networks where alternate formulas are used to compute probabilities. Here, we will label these techniques as "non-logical", even if a logic has been formulated for them, because these logics are very different to traditional approaches and rarely offer the same utility: properties such as completeness, termination, and even soundness may not exist. While these properties are attractive when reasoning about approaches, *e.g.*, to guarantee correctness or to prove complexity bounds, they need not necessarily hold for a system to be effective in practice. Furthermore, there is no reason to expect that these should hold for a good problem solver: they almost certainly do not for the most sophisticated computational system, the human mind.

2.2.1. Case-Based Reasoning

Case-based reasoning (CBR) is one of the earliest knowledge-rich problem solving techniques. By creating a library of specific case solutions, a CBR system can attempt to solve new problems by finding similar relevant cases and adapting the solution. As with psychological theories of analogy (§1.1.4), the important ingredients in these systems are:

> **Retrieval**: find an appropriate case from memory, typically by building an index on the problem goals or some analysis of problem type
>
> **Modification**: adapt the retrieved case so that it applies to the problem at hand (*cf.* mapping, §1.1.4)
>
> **Application**: if the adapted case can suggest a solution which transfers to the problem at hand, apply it to see whether it works (*cf.* transfer, §1.1.4)

Case-based reasoning has been successfully applied in many prominent domains, including law (Rissland & Ashley, 1987), and medicine (Koton, 1988).

In spite of this success, and in spite of its cognitive parallels, CBR is frequently excluded from the mainstream of AI research: recent textbooks give only a handful of pages, far less than for logic, productions systems, and search. One reason for this, in our view, is that there is no

standard technique for each of the three steps above. As the size and number of libraries can grow very large, good solutions for each of these steps becomes essential, but the solutions in the literature are varied and often specialised for certain applications. As a theoretical idea, then, CBR is inspiring but lacks the specificity and systematisability of the techniques in the previous section.

The most complex of these steps is retrieval. Although many different retrieval methods are used in current CBR systems, the two most common techniques are attempting to match on the problem goal or performing a look-up in a case index using certain dimensions (*e.g.*, keywords). If the goal is unique, or if the indices are relevant to the problem's "deep structure", this can lead to useful cases being retrieved, but as with human problem solving (§1.1.4) having the right index is frequently the key to expertise. In practice, CBR cases are often tagged by hand with certain pre-determined goal attributes or indices (or computed automatically on those dimensions, *e.g.*, using feature vectors described in §2.2.2.i). Finding the right tags for a problem domain and associating these tags with the cases can be a laborious task, and the retrieval methods are often not as flexible or extensible as users would like.

One retrieval method which is noteworthy for trying to address this shortcoming is the use of fuzzy "appropriateness conditions" for cases, introduced in the system Runner (Seifert *et al.*, 1994). These conditions resemble the formal preconditions of productions, but their formal applicability to a case is tempered by a computed utility of selecting that case (*cf.* success weighting, §2.1.5), and the result is used in building the case index. These "predictive features" display "greater success in recognizing when a past plan is in fact relevant in current processing" (*ibid.*); but in spite of its promising results, and the close correspondence to psychological theories, there has been relatively little follow-up on this work. Additionally, just as nearly all systems require developer-specified indices, Runner requires developer-specific appropriateness conditions, and so while it goes a long way towards making the best use of the retrieval indices provided, it has the same challenges to scalability and flexibility.

Another major obstacle to the uptake of CBR is the divergence — and often incompatibility — of CBR with the enormously powerful techniques of planning and iterative search. Production rules offer a means of describing abstract pieces of knowledge where applicability and consequences are clearly defined: such rules can be chained together and the result is a deterministic state space that systems can adeptly explore, in a narrow but thorough fashion, as a smart mouse might move through a maze. Cases, on the other hand, are usually difficult to decompose and difficult to abstract into general principles, but their richness provides many hints for assessing applicability and applying in diverse ways; consider an elephant faced with the same maze, who might remember it (as the proverbial elephant never forgets) or might pay it no mind and walk over it, but is not likely to explore its myriad twists and turns, backtracking eight steps and trying an alternate path. To return to gestalt terms, production rules encapsulate structural understanding at the smallest piecewise level, but without a holistic view; CBR goes some way to providing this holism, but at a cost of explicit deeper understanding. The use of appropriateness conditions has begun to merge the two, but still suffers from rigid, hand-coded retrieval methods and a strict distinction between this control knowledge and the actual domain knowledge.

2.2.2. Classification

Continuing a Gestalt analysis, we recall that the all-important "first step" in problem solving is "understanding the problem". In many analogical theories (§1.1.4), this is the crucial determinant of whether a solution is found: if a relevant concept is recognised, or a "deep" structural feature identified, it will help with the retrieval of useful prior knowledge; without this understanding, the utility of one's knowledge is purely aleatory. Although selection strategies have been described both for CBR (indexing) and the more logically structured techniques (production rules, search; §2.1.4), these strategies are usually *ad hoc* heuristics and nearly always inflexible. Remarkably, although the hardest part of a problem lies in choosing the right thing to do, the problem solving methods themselves (*e.g.*, search, production rules, CBR) do not inform this selection. As with the cognitive architectures in §1.2.3, these approaches (and systems) all require a developer to use pre-supplied control mechanisms not central to the theory or to write an *ad hoc* selection strategy.

Fortunately, a number of powerful techniques have been developed for classifying and retrieving knowledge. Although this has largely been in the context of either search engines or machine perception, we feel that its relevance to problem solving has been largely underrated, particularly in light of memory-based psychological theories in §1.1. Case-based reasoning is the one area which has most directly looked at using classification for retrieval, although there are others; we will review some of these hybrid systems in §2.2.4, after we review the techniques themselves.

Feature Vectors

Some of the earliest classification work associated a binary vector to entities, where each bit indicates the presence of a pre-defined feature in the entity. These "feature dimensions" can be placed in rank order of how well they partition the set of entities, or organised into a more complex structure. "Binary decision trees", a common such technique reminiscent of the game "twenty questions" and the taxonomy of species in biology, uses a hierarchy of features where the root node divides the set of known states as evenly as possible, into a group with the feature and a group without. For each group, the next most evenly dividing feature is found, and the result is a tree-like guide to questions which typically classifies all entities exhaustively, using only a tiny fraction of the potential features.

Many recent vector models allow "fuzzy features", where each feature's applicability to an entity is a real value in $[0, 1]$; techniques such as principal component analysis will identify a small number of dimensions (features or weighted groups of features) which optimally distinguish the entities. Entities whose feature vectors are similar after this dimensional reduction are likely to be similar, with regard to the space of entities; because the dimensional reduction focusses on the most relevant features, it will preserve important differences while removing irrelevant differences. Of

course, it is necessary for the important features to have been present in the first place, and a very large set of entities is needed for the results to be generalisable, but many domains have seen excellent results with this technique. Analysing language is a major example, where models such as latent semantic analysis (Landauer & Dumais, 1997) have analysed large corpora of text measuring the number of times a word X "co-occurs" in a passage near word Y, for all word pairs X and Y in some dictionary. This sets up a matrix of word entities (X) with a feature for each word it might co-occur with (Y): proponents argue that dimension reduction establishes a high-dimensional model of word meaning, with certain parallels to human cognition, and although this claim is contested, it is widely agreed that the technique achieves good performance in judging word synonyms and certain more sophisticated tasks in word semantics.

Neural Networks

One of the most celebrated innovations in AI has been the advent of neural network programming. Inspired by biological models of learning, this technique uses layers of nodes linked to form a network. An input vector from a training set is fed into the first layer of the network, and nodes in that layer "fire" signals of certain weights to nodes in the next layer. If an "activation" level of a node in the next layer passes a threshold, it fires to the subsequent layer, until eventually some output pattern is presented by the final layer in the network. The output pattern can be compared against the desired pattern for the input vector, and weights throughout the network are adjusted accordingly. Over repeated training instances, the neural network will — hopefully — converge to an arrangement where the right output is presented for each input. Provided this happens, and the training set is much larger than the size of the network, this convergence represents a general solution where relevant aspects of the input vector are being used to produce the output. If the network is given a new problem, not in the training set, it will apply the generalised solution to it, and assuming the training set was representative of the problem space, the output will correspond to the specific solution for this problem. Neural networks have been used successfully in many areas where conventional techniques have failed, often in classifying input as some canonical form, such as analysing handwritten digits or words, or recognising faces. (Kung, 1998)

Evolutionary Algorithms

Another metaphor which has shaped classification techniques is that of evolution. Code fragments can be treated like genes, with a large variety placed into a pool where they are mutated and combined to form programs which are then evaluated on some performance task. Those mutations and combinations which work well are kept from one generation to the next. After a great many iterations, good programs emerge from the process, with this "genetic programming" creating the solution from constituent pieces. This technique can be applied to synthesise algorithms for many purposes, but is particularly good for classifying data.

What unites all of these approaches, beyond their usefulness for finding categories or clusters of similarity, is their reliance on large sets of data. They do very well at finding explicit patterns of organisation, but they are all highly dependent on the input data and on the initial conditions, either the features chosen for the feature vectors, the topology of the neural network, or the starting set of code fragments. They do less well at finding subtle or highly structured patterns, and as with the techniques in §2.1.5, they can often attribute importance to accidental biases in the input data. These techniques are all essentially statistical engines, extremely powerful for finding classifications based on attributes, but rapidly weakened as the classification rules grow more complex.

2.2.3. Learning

As with the logical approaches in §2.1, the approaches described thus far in §2.2 all offer scope for automatically improving performance. Some of these learning areas are obvious — even implicit, such as a classifier "learning" the classification, or adding each new experience as a case in CBR — and others are more ingenious. Let us begin by noting that both availability and past performance, as described in §2.1.5, can be applied directly to CBR and to some of the classification techniques. The relevance of a pre-defined feature to some action (*e.g.*, selecting a category, or applying a specific case or rule) can be computed based on the frequency and recency of training exemplars where they are correlated. The learning techniques used to adjust neural networks often follow the same approach: a training set of inputs and outputs (*e.g.*, the desired classification) are provided, and firing patterns are reinforced if they lead from an input to the desired output. Intermediate ("hidden") layers in a neural network allow the learning of more complicated and subtle associations, and dimensionality reduction (*e.g.*, through principal component analysis) accomplishes a very similar function.

In a similar but more interesting vein, these correlations can sometimes be computed in an "unsupervised" setting. Some neural networks, given only a set of inputs, will "self-organise" so that the inputs are classed according to certain automatically found features; there is no guarantee this will be a useful classification, but surprisingly often it is (given a rich enough set of input data). If the classification is tied to an underlying action, such as, for example, using features to select cases or production rules, a system can automatically evaluate the applicability of the selected action and compute correlations appropriately. It should be noted that this formal applicability falls far short of indicating appropriateness; meaningful feedback on appropriateness, frequently, is only available after multiple sequential actions have been applied, making "credit assignment" (identifying the degree to which the appropriateness reflects on individual actions) difficult.

One other important area where learning has been applied to cases is the formation of generalised cases inductively (Muggleton, 1992). This technique has some relevance to the bottom-up learning of compound tactics (§2.1.5.ii), and some of the same mechanisms, but it focusses top-down on identifying common schematic solutions, from a number of cases. The

result might be a complete solution stripped of irrelevant detail and applicable to a wide range of problems; or it might be a fragment, akin to a tactical expression; or it might be a sketch, having lost some details essential to the individual solutions but providing a pattern which is particularly common, akin to subgoals in a plan. Using such a sketch for a problem requires filling in certain steps, but this can be easier than the original problem. Forming schematic cases in this way often leads to the same problem of too many generalisations as the inductive learning in §2.1.5.ii. There is one extra ingredient, however, which can help filter the results: detailed cases may include reasons why solutions work, and including these reasons in the abstraction, in explanation-based generalisation (§1.1.1.i), leads to fewer spurious inductions. Analysing the explanations to ensure the results are sensible requires yet more work (and inductive generalisation is time-intensive to begin with), and also relies on detailed semantics for cases, but in some specialised domains the technique has worked well (*ibid.*).

2.2.4. Hybrid Models

Much work has been done in AI which combines several of the approaches we have reviewed. Of particular interest to us are hybrid models which combine principled retrieval mechanisms (*e.g.*, classification) with powerful rule or case applications. Feature vectors, co-occurrence models, and neural networks have all been used in case-based reasoning to find potentially relevant cases for a problem at hand. A problem is first analysed with respect to the specified classification routine, and cases which are most similar, according to the classification, are retrieved.

A second interesting synergy is the use of associative techniques in ACT-R. A semantic network of "concepts" (usually keywords) is maintained and linked to productions. In a process of "spreading activation", concepts from a problem state or goal state — or from an element in "focus" — trigger other concepts in the semantic net, and this "contextual priming" leads to an increased likelihood of choosing associatively-linked productions.

As used in hybrid CBR and other types of hybrid models, however, the classification techniques capture only surface features of a problem. The retrieval mechanism is strictly separated from the structural case- or rule-based processing. As a consequence, the retrieval can neither be controlled through the use of cases and rules nor enriched by the results of this structural processing (apart from adjusting availability). A co-occurrence feature vector, for instance, would never distinguish between $\sin^2 x$ and $\sin x^2$, even if thousands of cases relied crucially on the structural difference; similarly, ACT-R's contextual priming would treat "sin" and "cos" as entirely different, even if thousands of productions were found to depend only on the presence of trigonometric functions. Thus retrieval in these hybrid models, as sophisticated as it is in comparison with other control heuristics, remains unable to recognise deep structural features. In Gestalt terms, they cannot do step one, "understand the problem", and without this, they cannot begin to do "productive thinking".

2.3. Theorem Proving

One domain we will focus on in depth in this thesis is proving mathematical theorems. This has been an attractive and active area of Artificial Intelligence for many reasons: it naturally permits formal specification; small formalisations can lead to a vast range of problems from simple ones, with short, known proofs, to tremendously difficult problems, with enormous proofs or even without any known proof; and success at automation will have many potential applications, from theoretical mathematics through to every discipline that relies on provably correct theorems. Indeed, one of the first published AI programs was Logic Theorist (Newell & Simon, 1956), as mentioned in §1.2.2, which used "inference rules" in first-order logic, tantamount to productions, to prove theorems automatically. Setting the stage for nearly all approaches since, Logic Theorist followed a hard-coded "executive routine" which, for a given proposition, attempts to apply the inference rules in certain pre-defined ways, in a heuristically-defined order.

Decision Procedures

Logic Theorist was an impressive success in its day, automatically proving many important theorems in first-order logic, but, in common with many such *ad hoc* control strategies, the success did not scale. Shortly after, the problem of theorem proving was solved, in a sense, by the discovery of "resolution theorem proving" (Robinson, 1965): resolution is a technique where a proposition is proven by showing that its negation leads to a contradiction; for many forms of logics, the technique can be implemented as a decision procedure guaranteed to solve the problem, often with guarantees on the time and memory required.

Decision procedures have been found for many types of problems in mathematics, from solving integrals and differential equations to checking arbitrary theorems in first-order logic. Computer algebra systems, such as Mathematica, Maple, and Matlab, typically have large libraries of such decision procedures and invoke them automatically, as necessary, to handle a user's request. These systems place a premium on computation expediency and ease of use, however, and these systems can sometimes fail to solve problems when a more configurable implementation might succeed. Furthermore, the emphasis on expediency means boundary conditions are sometimes overlooked, and computer algebra systems do sometimes make mistakes (§7.1.1).

Contemporary theorem provers are useful where a proof needs to be externally verifiable, or where more control over the solution attempt is necessary. There are several systems, such as Otter, NuPRL, PVS, Vampire, and SETHEO, which use heavily optimised decision procedures typically with an emphasis on high-performance theorem proving for first-order logic. These systems use techniques such as resolution (Robinson, 1965), "SAT" solving algorithms, and model checking, and are useful, for example, in mission-critical applications where high confidence is required for potential solutions.

Higher-Order Logics and Interactive Theorem Proving

One disadvantage of the high-performance first-order logic systems is that they are difficult to use for many areas of higher mathematics (*e.g.*, real analysis, number theory). First-order logic places constraints on notation which make writing such theorems unnatural. The proofs, consequently, are extremely complicated: nearly always too difficult for a human to understand, and frequently too immense for the systems to find automatically. An alternative, explored in the LCF (Milner, 1972; Gordon *et al.*, 1979) and HOL (Gordon & Melham, 1993) systems, is to use higher-order logics instead. Isabelle (Nipkow *et al.*, 2006), Omega (Melis *et al.*, 1999), Theorema (Buchberger *et al.*, 2006), λClam (Richardson *et al.*, 1998), Coq, PVS, and NuPRL are popular contemporary provers in this tradition: all offer increased expressiveness over first-order provers, although they lose some of the nice fully-automated properties.

Decision procedures are available in most of these systems, including many of the same techniques as in first-order provers. Resolution solvers are built-in to some of these systems, and others have capabilities to call to other provers or computer algebra systems. Theorema is built within a computer algebra system, Mathematica, and so the computer algebra capabilities are directly available. These systems all typically permit an author to write additional automation techniques, such as the simplification tactic of Isabelle (discussed in §4.2.1). Where automation or external systems are used, however, the validity of the result is only as good as the invoked technique: in some cases, the technique returns a proof which the prover can verify, but in other cases, the "proof" relies on the external technique being correct — such techniques are called "oracles".

For a large amount of the proving done in higher-order systems, however, there are no known fully-automated techniques which apply. Each of the systems mentioned in this section operates in an "interactive" mode, where the user can guide the system, suggesting automated techniques or specific lemmas from a library, step-by-step, until the goals are discharged. The result is a detailed, verifiable proof, but the process of getting there is not easy!

Proof Planning

Proof planning (Bundy, 1988) is one of the chief ideas for how to facilitate machine theorem proving when fully-automated decision procedures cannot be found. A schematic proof — or "plan" — records intermediate steps or states common to certain types of problems; analogous to some of the techniques in standard planning (§2.1.3), such plans can help determine how to break a complex theorem into sequences of lemmas which are more easily solved.

Although some effort has been made to synthesize proof plans automatically (Melis & Whittle, 1999; Duncan, in press), the most powerful plans remain those which are constructed by hand for certain domains (*e.g.*, rippling for induction, Dixon, 2006). Efforts to specify when proof plans should be used, *e.g.*, the Omega-Ants project, and how plans can be repaired, *e.g.*, critics

(Ireland *et al.*, 1999), have likewise mostly relied on *ad hoc* control strategies. These techniques have been shown to be very effective in the domains where they have been created, but they are subject to the limitations outlined in §2.1.4 and §2.1.5.ii. Whether expressed as decision procedures or as proof plans, control strategies — in all theorem provers — are currently written in a programming language at the level of the prover (usually tactics as in §2.1.4) rather than at the level of the knowledge base. In our view, this can make the control knowledge hard to understand; in practice, control strategy code is rarely re-used across domains (it is usually re-implemented) and rarely integrated with interactive theorem proving[17].

2.4. Conclusions

> The barrier that faced almost all AI research projects was that methods that sufficed for demonstrations on one or two simple examples turned out to fail miserably when tried out on wider selections of problems and on more difficult problems.
>
> (Russell & Norvig, 1995)

We have noted that real-world problems need not be tractable and can rarely be solved by scaling up solutions that worked for "toy problems" (contrary to popular expectation; §1.4.2.iii). The inapplicability of early AI approaches to handle these problems was one of the principal complaints cited in the Lighthill Report (1973), which was "highly critical of basic research in foundational areas [provoking] the so-called 'AI winter' " (Howe, 1994). The reaction has been a marked transformation to applied research and incremental advances to existing theories, and labelled as:

> a victory of the **neats** — those who think that AI theories should be grounded in mathematical rigor — over the **scruffies** — those who would rather try out lots of ideas, write some programs, and then assess what seems to be working.
>
> (Russell & Norvig, 1995)

"Both approaches are important," the authors continue, decrying the dearth of novel ideas. McCarthy also laments the narrow focus, where papers are "not directed to the identification and study of intellectual mechanisms" and "work only on theories that can be expressed mathematically in the present state of knowledge", and asserts a need for some scruffy work: "general purpose formalisms will be invented from time to time, and, most likely, one of them will eventually prove adequate" (McCarthy, 2000)[18].

The domain of mathematical theorem proving typifies the "neat triumph": huge libraries of theorems have been created (*cf.* expert systems, §2.1.1.ii), and powerful algorithms found for a large class of problems (using search, §2.1.2; proof planning, §2.1.3; and decision procedures,

---

[17] IsaPlanner (Dixon, 2006) is an interesting project in development to enable interactive or automated guidance of proof plans, but currently it operates only in its own interactive shell and does not have any facility for detecting applicability of plans.

[18] We in no way wish to claim that the formalism we will present is anywhere close to being proven adequate. Although we have a dream that — with a very great deal more work — it may do so, we will be very restrictive in this thesis about the actual claims we make. We provide the continuation of the quotation from McCarthy with wholehearted accord: "However, it would be a great relief to the rest of the workers in AI if the inventors of new general formalisms would express their hopes in a more guarded form than has sometimes been the case."

§2.1.4), yet even the most popular and powerful provers are, at the knowledge level, based around productions. The theoretical AI problem solving framework is the same as Logic Theorist — unchanged for fifty years — with the enrichment of domain knowledge (introduced by expert systems some forty years ago). It is subject to precisely the same critiques[19]:

> **(1) Difficulty in capturing "deep" knowledge of the problem domain.**
> **(2) Lack of robustness and flexibility.**
> **(3) Inability to provide deep explanations.**
> **(4) Difficulties in verification.**
> **(5) Little learning from experience.**
>
> (Luger, 2005)

The fundamental divide between the knowledge base and the control knowledge (§1.2.2) underscores all but one of these deficiencies[20], in our view. The importance of "structural features" in human problem solving (§1.1.1.iii; Wertheimer, 1945) could well translate to Artificial Intelligence, as their absence is consistent in psychological experiments (§1.1) with (1), (2), and (3), above, and their advent is postulated in corresponding theories as a key means for overcoming (5).

An interesting exploration, in our view, would be to attempt to use an extensible set of features to govern retrieval of cases or rules. One way of doing this is to remove the division between "domain knowledge" and "control knowledge", and extend the retrieval / application cycle presented in *e.g.*, §2.2.4: a system might first do a keyword analysis or look for common surface features; these might cue the system to search for "deeper" structural features; and then these deeper features cue an operator or case. Although structural features are typically more computationally expensive to detect, by treating them as domain knowledge, the system could attempt to restrict these tests to instances when they are merited. The selection of proof plans, productions, search strategies, or even evaluation functions could be controlled by this hierarchical execution strategy. One could imagine a system where a feature detector might recognise a game as chess, and so trigger a chess evaluation function, which might trigger a more specialised evaluation function, which would then cue a lookahead search analysis in a specific region, which then finally suggests the move. If the system is not given a chess board, none of these other steps will take place, so the system can maintain lots of knowledge and use it very efficiently. By using features at each step, the process may be made easier for a human to understand. It might even be possible to break down evaluation functions so that they compute solely with explicit, inspectable features.

---

[19] In the source text, these criticisms are applied to expert systems, but we do not think mainstream provers avoid their purview. An exception could be made for (4), as the main point of provers is the ability to verify a proof's correctness, but we note that verification has been achieved only for a tiny fraction of the algorithms used to find proofs.

[20] The exception, again, is (4). Interestingly, this difficulty is also characteristic of human thought, as shown by, *e.g.*, the Wason Selection Task (§1.2.1). In those domains where it is possible, verifiability of a result is usually sufficient.

Another attraction of problem solving in this way is that the distinction between cases and rules can be dropped. While it is far from certain that this is a useful thing to do, we note that combining techniques has been fruitful in many other areas, and for both cases and rules, there are types of knowledge for which one is preferable to the other. Following our terminology in Chapter 1, we will use "schema" to refer to cases or rules interchangeably, and also to other types of control knowledge such as search strategies, evaluation functions, or feature detectors. What emerges from this process is a generalised formulation of most of the problem solving approaches explored in this chapter: setting them in a single unifying framework is attractive, as is being able to relate it to cognitive theories. But what, if anything, does this generalisation buy us? The generalised formulation might be too vague to be useful (and that is why it explains production systems, planning, and CBR); or it might be an encompassing philosophical perspective that does not really change how anything works; or — possibly — it might suggest alternative ways of building problem solvers which may have new benefits. Given that most current problem solving systems ignore prior knowledge when guiding search or retrieval, and in the few instances where prior knowledge is used for retrieval, its use is severely restricted, we believe that this memory-based approach to AI problem solving has promise. As the scruffies point out, a question is interesting only when its answer is unknown: so let us explore whether this approach buys us anything new.

Chapter 3. Feasch — A System for Problem Solving with Features

We have seen in Chapter 1 that certain recent cognitive theories ascribe a primary role to features in the problem solving process. In Chapter 2 we showed that there is a wide variety of computational approaches to problem solving, with one of the main areas of difference being how each approach chooses when to apply various actions. These control strategies are summarised in Figure 3.1:

Technique	**Guidance**	**Result**
Production Rules	precondition	action
Search (simple)	search script (*e.g.*, depth, breadth)	action, production, or state
Heuristic Best-First Search	state-space evaluation function	state
Heuristic Local Search	action evaluation function	action or production
Planning	plan script	action or production
Tactics	tactic script	action, production, or tactic
Case-Based Reasoning	index	case

FIGURE 3.1: Control Techniques in AI

Case-Based Reasoning is the only area where explicit reference to "features" is standardly made, but there are two limitations we noted in §2.2.1: features in CBR are almost always restricted to easily identifiable surface attributes; and the retrieved cases can be difficult to decompose and use iteratively, *e.g.*, with planning and search. The ability to detect "deep" structural features[21] has been implicated as an important component of expertise (§1.1.2) but such features are rarely mentioned in an AI context. The approach of heuristic evaluation functions is most compatible with our idea, as the heuristic can attend to as complex a structural relation as the developer chooses, but the usual purpose of evaluation functions is to rank possible actions or productions. As such, it does not cue particular actions, but rather it evaluates all possibilities. Even when a heuristic is used to recommend particular actions, this cueing is integral to the heuristic; any feature it may note is incidental, and there is no scope in the theory for a separate process to

[21] We do not draw a strict, formal distinction between "surface" and "structural" features, but as is common in the literature we use these terms qualitatively. It is clear that detecting some features can require much more work — or more knowledge — than others. Internet search engines are very good at finding "surface" literal matches, *i.e.*, web pages where certain words or phrases are present. They can also offer "surface similarity" pages where related words or variant spellings are contained. However, it is a much more challenging and computationally intensive problem to find web pages where terms occur in a certain order, where a particular meaning of a term is intended (*e.g.*, "wind" a clock, not the "wind" blows), or where a set of relations holds with the re-use of one of more subjects (*e.g.*, Dan likes Daphne, but Daphne doesn't like Dan). Such "structural" features are not used in typical search engines. (The "semantic web" can be thought of as an example where structural features are explicitly marked up in data.) By stressing structural features for a system, we emphasise that systems should permit substantially more flexible definitions than would be afforded by literal matches or keyword similarity (see also §1.1.2).

inspect the partial result and trigger alternate actions. The same absence of features (or at best, their incidental inclusion) is true for productions, plans, tactics, and search strategies.

We identified a three-step problem solving process suggested by recent memory-based cognitive theories: features are detected in a problem space, these features cue schemas, and the most strongly cued schema is applied. There are similarities between this process and each of the techniques above — and it seems likely that with some minor changes, each would be compatible with the "feature cues schemas" process — but none of the techniques above have the same emphasis or distinctions as this three-step process. This leads us to pose a sequence of questions. Could this theorised three-step problem solving process be used to create a computational system, from the ground up? Assuming that it is possible, is it significantly different from other systems? Are there any situations where this approach might be a better choice for building a problem solving system?

3.1. The Possibility Hypothesis (PH)

Let us begin with the first point, and reserve the latter two questions for the next two parts of the thesis if appropriate. We wish to test what we will call "the possibility hypothesis":

(PH) Features can be the basis of an AI problem solving system.

As set out in the overview of this thesis, we take "features" to be representational units derived from a problem state recording information about the state and typically extending the provided description. We define them as "a known label or structured entity which can be associated with a problem state". We specifically include features which result from sophisticated analyses of the problem state, but *not* the analysis itself. A feature, as used here, is nothing more than a marker, such as a predicate expression; all actionable knowledge — including the means by which features are detected — is stored in schemas. By "basis" we mean that these features are an integral part of the system, *i.e.* problem solving cannot take place in the system without defining and using at least one feature; and by an AI problem solving system we mean a computational program which can provide an acceptable (under user-defined criteria) solution to a user-specified problem, optionally with the user defining control knowledge in the system.

The hypothesis says nothing about schemas, for a very good reason. We define a schema as a "knowledge element which describes an action which can be taken in a problem state"; consequently, it would be patently impossible for a problem solving system *not* to use this concept! Its identification as a top-level entity, however, we feel is important. Likewise, retrieval is not an explicit part of this hypothesis, but we note that the use of some such process is unavoidable. (In our formulation, knowledge retrieval is an action, and any retrieval process could be defined as a schema.)

The reader may observe that our definition of "AI problem solving system" includes programming languages; indeed languages such as C or Prolog are often used to develop problem

solvers for specific domains. The emphasis in (PH) is on requiring features to be a fundamental part of the system, and this is not the case for any programming languages or pre-existing AI systems of which we are aware.

Of course, even if a programming language fits our definition of an AI problem solving system, it may still be decidedly inappropriate. The TEX language, for instance, is well-suited for writing technical reports but would be a very poor choice for a chess-playing program — even though it is technically possible. Analogously, showing that features *can be* the basis of a problem solving system is not an especially noteworthy result, nor is it intended as such; further hypotheses must be proposed and tested before we can establish that such a system is significantly different from others or that it is appropriate for any class of problem solving tasks.

On the other hand, we might discover that features *cannot* be the basis of a problem solving system; this disconfirmation of (PH) *would* be noteworthy. Several accounts of human problem solving have posited that features play a key role. If we cannot build a computational system for problem solving based on features, we would have an argument that these cognitive theories are under-specified at best, and flawed at worst. Our interest in (PH) is merely to see whether we can exclude this possibility before proceeding to the other questions above, and if we cannot, ideally to identify concrete obstacles that prevent the abstract theories from becoming implementable models.

3.2. The Architecture

We will test the possibility hypothesis by attempting to develop a problem solving system following the feature-based approach: given a problem to solve, the system will first search for specified features; these features will then cue schemas for problem solving; and then these schemas will be applied to the problem, either solving it, transforming it to a new problem state, or leaving it unchanged. Given the primacy of features and schemas, we call our architecture "Feasch". (Both terms are used in this section as defined in §3.1; commonly, schemas will be operators as in (2) below, although the definition includes feature detectors as in (4) below and other as yet unspecified entities.)

In defining the specifications for our feature-based system, let us begin with the components essential to any AI problem solving system (irrespective of the use of features):

(1) a description of a problem state

(2) a set of operators which act on problem states in a given problem domain (or a means of generating such such operators)

(3) a means of choosing an operation from (2) for a given problem state (1)

Examples of these components were shown in Chapter 2; Newell & Simon's Generalised Problem Solver, for example, defines (1) in a predicate logic and (2) as production rules. Component (3) is the most complicated: it records the "control knowledge" which determines how the problem

solving system actually runs. In GPS, (3) is also encoded as production rules, where preconditions are tested on a problem domain and then determine the operator to apply.

In feature-based problem solving, component (3) will be responsible for identifying features in a problem state and then cueing schemas from these features. A key difference between this approach and that of production rules can be observed here, that features are found and explicitly recorded *independently* of selecting the operator. (This is not directly relevant to testing (PH), but the question of the uniqueness of this approach — and the identification of differing behaviour or capabilities — will become important in latter parts of this thesis.) The explicit step of identifying features introduces two more essential components:

(4) a set of detectors which determine the presence of certain features within a problem state in a given problem domain

(5) a place for recording derived information about the problem state

and the use of features to cue schemas implies the following expansion of component (3):

(3a) a description of how a feature detected by (4) cues *schemas*, by which we usually mean operators from (2) but might include detectors from (4) or other, yet unspecified, entities

(3b) a means of choosing a *schema* (such as an operation from (2)) given cues from (3a) and derived information from (5)

We note that these six components can be divided into two groups. Some will be static across an entire problem area — that is, they do not change between problems in the same domain — whereas others must be defined for each individual problem. From an end-user perspective, it therefore makes sense to group the components into two general containers:

(6) a problem domain definition, including (2), (4), and (3a)

(7) a problem context, including (1), (5), and (3b), as well as a reference to (6)

The remainder of this section turns our attention to implementing the architecture in software; it can be skipped, along with subsequent sections labelled "Implementation", by a reader whose interest in programming or technical details is minimal.

3.2.1. Specification

Our first design choice was to define abstract high-level interfaces for features and schemas, **`IFeature`** and **`ISchema`**. These interfaces are empty, apart from bookkeeping fields and methods for things like `name`, `id`, and a pretty-printing `toString` representation; nevertheless, they serve a number of useful purposes. On a theoretical level, they clarify the implementation's connection to the cognitive theory; on a design pattern level, they support an abstract grouping of related entities (such as components (2) and (4)) and allow methods to specify the type of object(s) they expect in an extensible way; and on a object-oriented programming level, they enable the system to ensure that features and schemas are used appropriately through static type identification. Simple implementations for these objects will be presented shortly, and of

course a programmer can extend and implement these interfaces as necessary for any particular application. In accordance with good modular design, such an extension will require few, if any, changes to the rest of the system.

Beyond these two interfaces, we have attempted to maintain a close correspondence between the eight logical components identified and the formal classes used in the system. This is done through the following eight interfaces with method signatures as below (excluding non-essential fields and methods such as `name`, `id`, and `toString`):

(1) `interface` **`IProblemState`**

`Object get()`: returns a representation of the problem state (the programmer can choose type of object depending on the domain; commonly it is a string or predicate expression)

(2) `interface` **`IOperatorSchema`** `extends ISchema`

`IProblemState run(IProblemState)`: transforms the given problem state, returning a new problem (or throwing an error if the operator cannot apply)

(3a) `interface` **`IMapping`**

This interface is empty: `IExecutor` implementations will specify sub-interfaces of `IMapping` depending on the requirements of a particular execution strategy

(3b) `interface` **`IExecutor`** `extends ISchema`

`void onFeatureFound(IFeature, IMapping)`: applies the mapping for a found feature, looking up the `IMapping` from the `IProblemDomain` if null, and then cues the mapped schemas if appropriate

`void onNodeCued(ISchema, Object)`: performs the appropriate action when a schema is cued (*e.g.* from a mapping on a found feature); this might be to run the schema if it is a feature detector, or inform `IProblemContext` if it is an operator; the second parameter passes optional extra cue data

— `{start,run,suspend,resume,getStatus}()`: standard thread methods

(4) `interface` **`IFeatureDetector`** `extends ISchema`

`void run(IProblemContext)`: tests for the applicability of one or more features in the context's state, calling `IProblemContext.onFeatureFound` when a feature is found

(5) `interface` **`IFeatureBlackboard`**

`void put(IFeature, Object)`: adds a feature to the blackboard, along with an optional second piece of information

`Object get(IFeature)`: returns the second piece of information associated with a feature on the blackboard, or `null` if the feature is not on the blackboard

`List<IFeature> getFeatures(IFeature)`: returns a list of features on the blackboard which match the passed feature; everything if the argument is null, the single feature if there is an exact match, or an empty list if none match; some implementations may support wildcard specifications

(6) `interface` **`IProblemDomain`**

`List<ISchema> getAllSchemas()`: returns the list schemas for this domain

`List<—> get{Operators,Detectors}()`: as above for common schema types

(7) `interface` **`IProblemContext`**

— `getProblem{State,Domain,Blackboard}()`: accessors for *(1)*, *(5)*, and *(6)*

`void onFeatureFound(IFeature, IMapping)`: called by feature detectors; adds the feature to the blackboard and informs listeners

— `{add,remove,get,clear}FeatureFoundListener[s](—)`: usual pattern for registering classes interested in when new features are found; typically the only listener is an `IExecutor`, but theoretically it could be anything that has an `onFeatureFound` method

`void onOperatorSchemaCued(IOperatorSchema)`: called from `IExecutor` when an operator is cued; typically broadcasts event to registered listeners

— {add,remove,get,clear}OperatorCuedListener[s](—): usual pattern for registering classes interested in when operator schemas are cued; often a user's class will listen for these events, or sometimes there are no listeners

Classes implementing each of these interfaces will be described shortly. For now, simply given these interfaces, the flow of execution can be described as in figure 3.2.

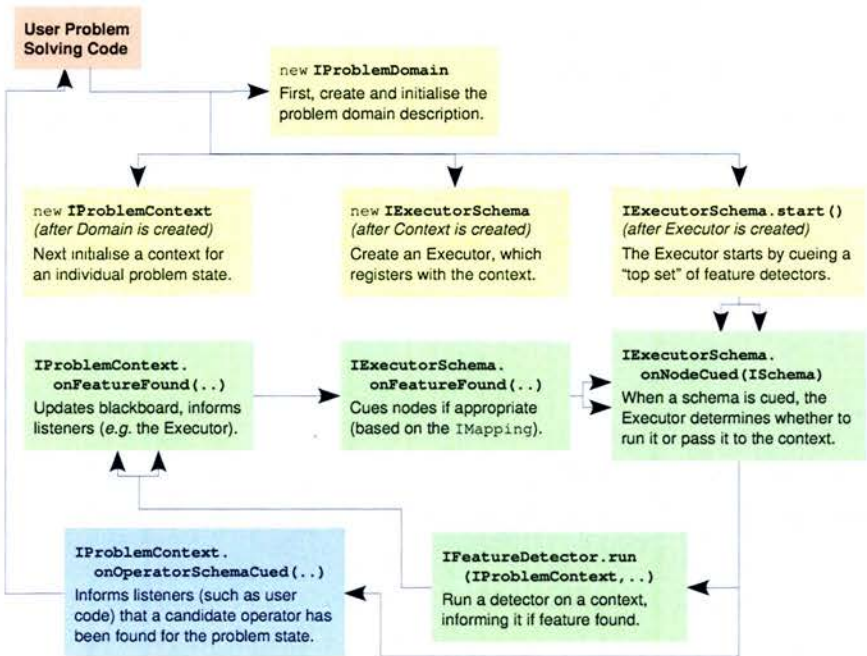

FIGURE 3.2: **Execution Flow in Feasch**. This diagram shows a typical execution flow for problem solving with the Feasch architecture. Arrows indicate which methods call to other methods: ordering requirements are shown in italics, and conditions on the calls are described in the boxes. A pair of arrows leading to a box indicates any number of calls, depending on the situation, and possibly made in parallel. For ease of reading, initialisation is shown in yellow, schema-based problem solving activity in green, and output candidate operators in blue.

To perform problem solving with the Feasch Architecture in a particular domain, the user must define the relevant `IFeatureDetector` and `IOperatorSchema` instances, and optionally `IProblem{Domain,Context,State}` implementations. For a given problem, a user's code will typically first create the `IProblemDomain` and initialise it with the relevant schemas. The user's code will then create the `IProblemContext`, initialised with the `IProblemDomain` and with the particular `IProblemState` being considered.

An `IExecutor` is then created to control how and when features cue schemas. A variety of standard implementations are available — representing different execution strategies — and are described in §3.3. The `IExecutor` should be linked with the `IProblemContext` where it will run, and it may also be initialised with a set of `IFeatureDetectors`. (Alternatively, these can be obtained from the `IProblemDomain`.) The feature-schema loop then begins by a call to `IExecutor.start()`, which cues each of the initial feature detectors specified (either sequentially or in parallel). In response to these cues, the `IExecutor` simply calls the `run` method on those `IFeatureDetectors`, which tests for the presence of features. For each feature found, the `IProblemContext` is notified; the `IProblemContext` then notifies the `IExecutor`, which applies the mapping to schemas implied

by this feature (again, several implementations are available, discussed in in §3.3). Each of the schemas cued by a feature is then considered by the `IExecutor`: if it is of type `IFeatureDetector`, it is run following the same cycle as just described.

The other situation that may arise is that the cued schema is an `IOperatorSchema` instead. This means that the architecture has identified a suitable candidate operator for the problem, on the basis of the features and feature-schema mappings. When this happens, the `IExecutor` informs the `IProblemContext`, which will typically call back to the user's code. Unless the user's code specifies otherwise, the `IExecutor` continues to run the cued feature detectors (and informing the `IProblemContext` of the cued operators) until no further features can be found. In a typical situation, however, the user's code might suspend the `IExecutor` when a candidate operator is found, and apply `IOperatorSchema.run()` on the problem state. This may fail, in which case the user's code would probably resume the `IExecutor`, or else it results in a new `IProblemState`. If the `IProblemState` represents a solution, the problem solving task is completed; if not, the user's code would typically create new `IProblemContext` and `IExecutor` objects and repeat the entire process on the new state.

3.2.2. Implementation

An advantage of defining the specifications as interfaces is that multiple implementations can be provided for different purposes. We describe standard implementations of all interfaces except the `IExecutor`/`IMapping` pair in this section, along with the key design choices. A set of progressively richer `IExecutor`/`IMapping` strategies is described in the next section along with a sample application of the architecture.

Language and Environment

We have chosen to develop the system in Java because it offers robust support for object-oriented design, allowing us to use the essential interfaces just identified, make them public, and create our implementations separately. This simplifies the task for a user of our system, since she need not be aware of our implementations, but also allows her, if necessary, to create her own alternative implementations. Furthermore, Java has mature handling of concurrency, which we expect will be a strong advantage in cases where multiple schemas, especially feature detectors, have to run in parallel. Good networking support and portability, as evidenced by the large number of cross-platform Java web applications, are further reasons for our choice of programming language.

Specifically we use the Sun Java 1.5.0 SDK (standard edition), which is the industry-standard, because of its widespread adoption, proven stability, and interoperability features with Java 1.4. The only 1.5-specific feature is the use of generics, both in the notation in this chapter and in the code; this makes code more flexible and readable, and as it can be used at compile-time to generate code compatible with Java 1.4, it does not introduce interoperability difficulties.

All development described in this thesis has been done primarily under Linux, with unit testing in the JUnit framework under Windows XP and Mac OS X as well as Linux. The Eclipse IDE v3.2 was used in development, and its JAR libraries are needed in those instances where our system is run with Eclipse graphics (*e.g.* the Eclipse plug-in we present in §4.3.4). For concurrent sorted maps, we have used classes from JSR 166 which are expected to be included in future versions of Java. Unless specifically indicated otherwise, no other third-party libraries or packages have been used and all code is © 2007 by Alex Heneveld with all rights reserved.

Classes

To simplify the creation of a Feasch model, we provide some standard general-purpose implementations of all the interfaces described in the previous section. The definition of features and schemas, of course, will depend on the problem area, so we provide implementations of `IFeature` and `ISchema` which facilitate their domain-specific instantiations.

As predicate representations are useful throughout many AI systems, we have chosen to use this style in the `SimpleFeature` implementation of `IFeature`. This class's constructor takes a `String` name followed by zero or more `Object` arguments (which may be other features). The `toString` method aids in pretty-printing such a feature, and comparators are provided for sorting these features by name then by arguments. The class `FeatureDictionary` implements `IFeatureBlackboard` and uses this comparator for tree-based retrieval of features.

On the schema-side, an abstract superclass **`BasicSchema`** fulfills most of the contract methods required by `ISchema`. This class takes a `String name` and an optional `String identifier` in its constructor, and provides the relevant accessor methods. (If the `identifier` is not supplied, a unique random ID is chosen.) By extending this class, operator schemas and feature detectors can be defined simply by implementing the relevant `run` method. By way of example, we might implement a feature detector for whether the state is an even number as follows:

```
public class EvenNumberDetector extends BasicSchema
   implements IFeatureDetector {

  public EvenNumberDetector() {
    super("even-number-detector");
  }

  public void run(IProblemContext context) {
    if (context.getProblemState().get() instanceof Number) {
      double d = ((Number)(context.getProblemState().get())).doubleValue();
      if (d/2 == Math.floor(d/2))
        context.onFeatureFound(this,
           new SimpleFeature("even-number", (int)d), null);
    }
  }

}
```

If the problem state is a number representing 6.5 or -3, or if it is not a number; this detector does nothing. If, however, the problem state is a number representing, say 200, this detector informs the context that a predicate feature `even-number(200)` is found.

For the purpose of problem solving in a particular problem domain, the relevant schemas are usually set in an `IProblemDomain`. A general-purpose implementation of this interface is provided in **`BasicDomain`**, using a `List<ISchema>` object for storing schemas and a `Map<String,IMapping>` object for storing default mappings by `String` feature name. Schemas can be added to this list through the `addSchema(ISchema)` method; the accessor methods `getOperators()` and `getDetectors()` provide filtered views of this list for `IOperatorSchemas` and `IFeatureDetectors` respectively. Default mappings for features can be specified by the `addMapping(IFeature, IMapping)` and `putMapping(IFeature, IMapping)` methods and accessed by the `getMapForFeature(ISchema, IFeature)` method. The class's constructor takes a `String name` which describes the domain.

Within a problem domain, an individual problem is recorded as an `IProblemContext`; the class **`BasicContext`** provides a standard implementation of this interface. The constructor for this class takes a domain-specific `IProblemState` (a simple generics-based implementation is provided in `BasicProblemState<E>`) and a reference to the `IProblemDomain`, which are accessed by `getProblemState()` and `getDomain()` respectively. The class uses an `IFeatureBlackboard` (in the class **`FeatureDictionary`**) which supplies a comparator for features of the form `predicate(arg1, arg2, ..., argN)` and stores features in a `TreeMap` for efficient lookup with wildcard support. The `onFeatureFound` method in this class adds features to this blackboard (called by the `IExecutor` in the standard model), and the `getBlackboard()` method lets a feature detector or other schema inspect the set of features found for this context. The class also provides support for adding listeners for found features and cued operators, and handles calling to these listeners when features are found or operators are cued.

The execution of feature detector schemas and the cueing of operators is handled by an `IExecutor` implementation. Different implementations, including different mapping strategies, may be suitable for different situations; we present three such implementations in the next section. These implementations — and, conceivably, others — inherit from an abstract superclass `BasicExecutor` which provides lifecycle methods, including `suspend()` and `resume()`, and a boilerplate `onNodeCued(ISchema s, ..)` which dispatches calls to `on{OperatorSchema,FeatureDetector,UnknownSchema}Cued` as appropriate. In this superclass, each of these three methods simply logs the cue; it is intended that this class be extended by a developer's overriding these `on...Cued` methods as appropriate (executor-specific mappings are passed in a parameter to these methods). The developer should also implement the abstract `runSingle()` method, whose contract is to iteratively run the single most appropriate schema. Together, these methods can define an execution strategy.

Design Efficiency

Many problem solving tasks require a search through a large state space, so in designing the architecture we have concentrated on efficiently supporting the creation of a large number of contexts and the execution of a large number of schemas. Furthermore, the architecture will likely be used in standalone problem-solving situations where speed is of the essence, as well as interactive environments where control and feedback are important.

The separation of a problem domain from the context is a particularly important enabler of efficiency: this allows a state-specific `IProblemContext` object to be lightweight (*i.e.* they can be created in large quantity with low overhead) whilst having access to the set of operators and feature detectors relevant to the problem area.

The use of listeners registered with the context allows support for interactive application, by adding listeners which update the user interface, and high-throughput standalone applications, by adding a single listener which determines whether to stop feature detection and run a cued operator schema.

The `IExecutors` support `suspend` and `resume`, so applications can control the system resources depending on their needs. Applications can rapidly switch focus among problem states by invoking these lifecycle methods on the `IExecutor` for a particular `IProblemContext`, or maintain several running in parallel for a best-first feature-detection strategy across multiple states. Interactive applications, in particular, can use guidance from the end user to constrain feature-detection to those areas which the user advises without restricting the efficiency of using Feasch for fully-automated problem solving. Furthermore, the `IProblemContext` and `IExecutor` implementations that we use both support a `clear()` method for when a problem state is no longer active, so that the executor can be re-used for a different state or context without the overhead of re-creating and initialising.

On a technical level, thread-pools are used where possible to optimise thread creation, and the specification of formal interfaces for `IFeatureDetector` and `IOperatorSchema` means that the expensive dynamic run-time type identification call `Class.isInstance` can be replaced by the much more efficient `instanceof` check.

3.3. Execution Strategies: Testing (PH) on Noughts-and-Crosses

We will now turn our attention to exploring whether Feasch can actually be used for problem solving, testing the possibility hypothesis (PH) on a rudimentary two-player game. Before we describe the game and the domain-specific strategies, however, we must introduce the general execution strategies available in the system.

As yet, not much has been said about the `IExecutor/IMapping` strategies available (3a and 3b from the list in 3.2.1). The primary reason for this is that the choice of execution strategy is an implementational detail not specified in the underlying theory of how features cue schemas:

although some strategy is essential for problem solving with features and schemas, there is an essentially limitless range of possible strategies. As part of Feasch, we include three formalised execution strategies, representing progressively richer functionality based on our consideration of a number of problem solving tasks and several pre-existing AI techniques.

- **`ListExecutor`** runs each feature detector schema sequentially from a fixed list, with each found feature cueing a list of any number of operator schemas; listeners are informed on each cue to an operator schema
- **`WeightListExecutor`** runs detectors sequentially from a fixed list, but with weights permitted in the map from features to a list of operator schemas; listeners can at any point access a list of cued schemas ordered by total weight
- **`NetworkWeightListExecutor`** runs the highest-weighted detectors (possibly in parallel) from a dynamic list, with features giving a weight cue to operators or to other detectors

The first of these is the simplest strategy, and each subsequent `IExecutor` represents a strict increase in capabilities. They are most easily understood in context, so in this section we describe them with reference to a particular domain, one by one, building up to the synthesis of all capabilities in `NetworkWeightListExecutor`. This strategy, the most sophisticated, is suggested as the best default choice and, except where explicitly noted to the contrary, is used for the experimentation in this thesis.

It should be stressed that the Feasch architecture will work with any implementation of the `IExecutor`/`IMapping` interfaces, and while the ones we outline here represent our best effort, they should not be seen as a limitation on what might be possible. In our exploration of the possibility hypothesis, one issue we will have to address in the event of impossibility is the extent to which this reflects on our implementation; consequently, in this section we will attempt to make the case that the `NetworkWeightListExecutor` strategy (§3.3.3) represents a good combination of desired functionality, and that if it has serious limitations, this in fact does raise important challenges for the underlying theory to be more explicit about how execution happens.

The initial domain where we will test (PH) is the familiar game "noughts-and-crosses", also known as "tic-tac-toe". The game is played on a 3x3 grid with two players, using tokens `X` and `O` respectively, who take turns placing one of their tokens in any empty square of the grid. The first player to place three of his tokens in a line — horizontal, vertical, or diagonal — is the winner. An example game is shown in Figure 3.3.

FIGURE 3.3: A Sample Game of "Noughts-and-Crosses"

The game itself is trivial, in the sense that there are many strategies for perfect play which are not difficult to find. For our purposes, this makes it a good initial test case. If our architecture allows us to encode these strategies through features and schemas, and play well with some

execution strategy, the possibility hypothesis will be confirmed for this example[22]. If, on the other hand, some strategies cannot be expressed (or are extremely cumbersome to express), we potentially have an interesting disconfirmation of (PH). We emphasise again that in this experiment, a disconfirmation would be the more interesting result, for it would imply that the cognitive theory is under-specified. A confirmation is less interesting since the domain is so simple, and would merely serve to justify further exploration in more sophisticated domains. The actual experiment is described in §3.3.3, using the most advanced executor; explorations with the other executors are described first, to give understanding about the execution strategy and the approach. Indeed, beyond merely testing the possibility hypothesis, these explorations will be used to collect anecdotal evidence about how features and schemas can express problem solving strategies. In the event that (PH) is confirmed for noughts-and-crosses, these qualitative observations will be useful for formulating further hypotheses for testing in more complex domains.

3.3.1. A Simple List Executor Strategy

Let us begin with the following simple strategy: if any line has two identical pieces and a blank, play there, preferring lines with my tokens; otherwise, prefer the centre, and prefer corners over other squares. The `ListExecutor`, with its simple strategy of running detectors in order and cueing operators directly, can be used with the strategy encoded as follows:

(1) run the `i-can-win` detector
(2) run the `opponent-can-win` detector
(3) run the `centre-free` detector
(4) run the `corner-free` detector
(5) run the `possible-move` detector

All features indicate a square on the board and map a cue to the corresponding `play-at` *operator.*

The `ListExecutor` runs these detectors in order; a successful detection informs the context of the feature (or features) found, and here every feature cues a single schema. In our implementation, the executor stops running when the first operator is cued, and that operator is applied to the current game (or displayed on the screen).

To test whether the Feasch architecture successfully captures the noughts-and-crosses strategies described, we will implement these models and run them against a purely random player. With more complicated strategies (and the final `NetworkWeightListExecutor`), we would expect perfect play, but in this simple example we will count a success if the model performs significantly better than chance.

---

[22] A note on the scientific validity of these experiments: the three execution strategies were developed without consideration of noughts-and-crosses, and the results described in this section were obtained without altering the executor classes for this domain.

Implementation — Noughts-and-Crosses Test Platform

In implementing our tests, we use various `Player` classes to control the different strategies in the Feasch architecture; the simplest of these is `RandomPlayer`, who uses the standard Feasch `CueOpsRandom` detector; this detector randomly chooses one or more operator schemas O from the domain and reports a feature `random-cue(`O`)` with a mapping to O. `ListMain.ListPlayer` uses the detectors described above, with `CueOpsRandom` used as the fall-through to ensure all possible moves get a cue if no other features are found.

The `Players` implement a `IOperatorSchema run(T3Board)` method, which applies the relevant Feasch model to a given game state (stored in the class `T3Board`). A `Game` class tracks progress on an initially empty board, calling each player's `run` method in turn until the game has a winner (or is a draw). The `Games` can be run in verbose mode, to explore what is happening, or in batch mode for high-volume testing (output is suppressed, apart from summary totals, and the first move alternates between players for fairness). These classes, and all the execution strategies for noughts-and-crosses, are contained in a package `org.heneveld.feasch.tictactoe`.

Implementation — The List Executor

Formally, the `ListExecutor` is an implementation of `IExecutor` inheriting from `BasicExecutor` (described at the end of 3.2.2.ii). It is instantiated with the `IProblemContext` instance describing the problem where it will run, and then feature detectors are added to the `List<IFeatureDetector>` returned by the `getQueuedDetectors()` method (or all detectors in the context's domain can be added by calling `initFromDomain()`). For each feature, a list mapping (in `SchemaListMapping`) can specify which operator schemas should be cued when the feature is found; these `IMappings` can either be static (set by the domain) or dynamic (computed by the detector). By registering `Listener` pattern objects on the executor, a caller can be informed whenever a feature is found (`IFeatureFoundListener`) or whenever an operator schema is cued (`IOperatorSchemaCued`). Once this initialisation is complete, the list executor's `run()` method will cycle through the feature detectors, looking for features and cued operators. The following code (from the class `RandomExample`) demonstrates the use of a single "random" detector operating with an `IOperatorCuedListener` that records operator schemas in the order they are cued.

```
IProblemDomain domain = new NoughtsAndCrossesDomain(
   "noughts-and-crosses-p1", T3Board.X);
BasicContext context = new BasicContext(
   new BasicProblemState<T3Board>(board), domain);
final List<IOperatorSchema> ops = new ArrayList<IOperatorSchema>();
context.addOperatorCuedListener(new IOperatorCuedListener() {
  public void onOperatorSchemaCued(IOperatorSchema op, Object message,
     Object param) {
    ops.add(op);
  }
} );
ListExecutor executor = new ListExecutor(context);
```

```
executor.getQueuedDetectors().add(new CueOpsRandomList(CueOpsRandomList.ALL));
executor.run();

System.out.println("found "+ops.size()+" candidate operators:");
for (IOperatorSchema op : ops) System.out.println("  "+op);
```

A few details on using the `ListExecutor` should be noted. It is not permitted for features to cue anything other than `IOperatorSchemas`, and the optional `param` field in the `IMapping` cue is ignored by the executor. Once the executor starts, new `IFeatureDetector` schemas cannot be added to the queued detectors list. In summary, the `ListExecutor` is confined to simple domains using a static list of detectors whose features send a binary cue to operators. (Other executors overcome these limits, but the `ListExecutor`, being the simplest execution strategy, is easiest to understand initially.)

The noughts-and-crosses strategy described above fits with the `ListExecutor`, and the code used in `ListPlayer` is very similar to the code above, with two differences. The method `executor.suspend()` is invoked whenever an operator is found, and it is not `resumed` unless the operator fails on the current problem state. Also, of course, a number of other detectors are added to the executor's "queued detectors" list, in accordance with the strategy described above. Each of these are implemented in classes contained in `ListMain` which implement `IFeatureDetector` and extend `BasicSchema`, and each detector tests for the presence of an eponymous feature with the indicated argument(s).

- **`i-can-win ("`L`", "`x`,`y`")`**: examines all possible lines to see if the current player can win there on the next turn; returns L as a description of the line, with x and y giving the absolute co-ordinates of where to play
- **`opponent-can-win ("`L`", "`x`,`y`")`**: examines all possible lines to see if the opposing player could win on the next turn; arguments are as for `ICanWin`
- **`centre-free ("`x`,`y`")`**: checks if the centre square is empty, detecting a feature with arguments `1,1` if so
- **`corner-free ("`x`,`y`")`**: checks whether each corner square is empty, detecting a feature with appropriate coordinates for each empty corner

The `IOperatorSchema PlayAt` is used to add the player's token to the board (also in the `tictactoe` package). The domain is configured a `FeatureToMoveMapping` object for each of the features above; when a new feature is found, the executor retrieves this object from the domain, its `cue` method tells the executor about the cue to the relevant `play-at ("`x`,`y`")` operator, and the `ListExecutor` passes this cue to the registered `IOperatorCuedListeners`.

Results

A test battery of one million games sees the simple strategy just described winning 90.41% of its games against a random player, drawing 8.99%, and losing 0.60% (margin of error <0.029%); for reference, in a contest of two random players each player wins 43.66%, with 12.67% of games ending in a draw ($\pm$0.093%).

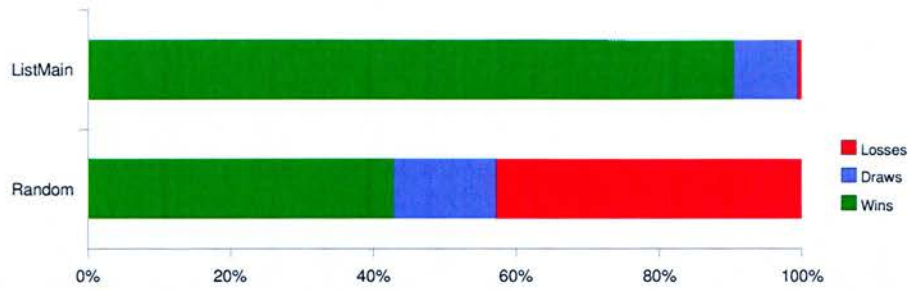

FIGURE 3.4: Performance of List Player against Random

This simple strategy significantly outperforms the chance player. Furthermore, it was straightforward to code the strategy in the Feasch architecture, and powerful performance was achieved without much domain-specific code. More complex strategies, borne out of the 0.6% of games that were lost, will prove a better experiment for (PH), and so we delay discussion of this experiment until §3.3.3.

The `ListExecutor`, since it only affords simple ordered "if-then" style processing, looks a lot like production rules. It has the attendant benefits that the knowledge can be decomposed, pieces evaluated independently, and alternative strategies examined easily: by removing some of the mappings, for example, we find that ignoring the `corner-free` feature results in the loss percentage rising to 5%, whereas responding to the `i-can-win` feature alone accounts for 80% of the wins. A possible difference to production rules already emerges, in this languaging: the "preconditions" are encoded independently of the mapping, and so we can speak of "acting on" or "ignoring" a feature, and the model makes this explicit. Casually speaking, the Feasch model seems to mirror natural ways of describing problem solving strategies, at least for this example.

Let us explore whether Feasch can be used for more complicated strategies; in particular, since noughts-and-crosses is not difficult to solve, we would expect to be able to encode a perfect-play strategy in our architecture. Most of the games which were lost followed a pattern where the opponent played first and then our strategy set up two of our tokens in a line such that the opponent, by blocking, set up two separate plays where he can win. At that point, even though our strategy blocks one win, the opponent can still win at the other. We could attempt to correct for this by coding separate features for all the known losing cases, but it would be far preferable if we could use a heuristic that combines features when cueing operators, to prefer squares satisfying certain conditions. The `ListExecutor` does not permit this, so we turn our attention to the next execution strategy.

3.3.2. Weighted Lists

One limitation on the `List` executor is that the cues are all-or-nothing; a corner being free can cue a move in that corner, but the only way to specify that the centre is better to constrain the order in which feature detectors are run. There is also no way to apply cues additively; that is, if two features cue the same schema, it would be nice if that schema is then preferred to others.

The `WeightList` executor overcomes these limits by using numeric weights as cue mapping a feature to a schema. Cued operator schemas are ranked by the sum of the weights that cued them, and at the end of a run—or when an operator has a strong enough cue—the operators can be presented to the user or tried in order, as in local best-first search.

For noughts-and-crosses, we could use the following weights to achieve behaviour similar to the `List` executor, but without relying on the order in which detectors are run:

- **`i-can-win`**`("`L`", "`x`,`y`")` cues `play-at-`x`-`y with weight **2.0**
- **`opponent-can-win`**`("`L`", "`x`,`y`")`: cues `play-at-`x`-`y with weight **2.0**
- **`centre-free`**`("`x`,`y`")`: cues `play-at-1-1` with weight **0.5**
- **`corner-free`**`("`x`,`y`")`: cues `play-at-`x`-`y with weight **0.4**

Some additional features which could be used to discourage the losing cases observed in the last section are `next-to-opponent`, which notes that an empty square is adjacent to an opponent's piece, and `in-wasted-line`, which detects if a square is in a line which already has an `X` and an `O` (and so that line cannot be a win for either player even before our play). From observation, we've discovered that it is good to play near an opponent's piece, particularly adjacent along the diagonal, except when the square is part of a "wasted line". For this example, we set the following weights:

- **`next-to-opponent`**`("`x`,`y`")` cues `play-at-`x`-`y, with weight **0.4** if it is orthogonally adjacent, and weight **0.6** if it is diagonally adjacent
- **`in-wasted-line`**`("`L`", "`x`,`y`")`: cues `play-at-`x`-`y with weight **-0.3**

As the reader may infer, negative weights are used to discourage the target schema. The weights are effectively additive, so a corner square next to an opponent's piece would get the same cue as a single feature mapping a weight of 0.8, and an edge-middle square in a line with an opponent's piece and one of my own pieces would get an effective cue of 0.1. However, integer weights are treated specially, and a cue of 1.0 is stronger than any number of repeated cues of 0.5: this allows us to encode that blocking an opponent is always more important than strategy such as playing at the corners. (Similarly, a cue of 2.0 is always stronger than repeated cues $|\delta| < 2$, so a play which wins for us is preferred to a play which blocks an opponent. Full details of how the weights are added is described in the next section.)

Implementation

The class `WeightListMain` in the `tictactoe` package records the features and execution strategy used for the model just described. The main difference between this code and the `ListMain` code shown previously is that the `IMapping` implementation, `SchemaWithWeight`, sends a `double`-value "weight" as the parameter when cueing a schema. The `WeightListExecutor` maintains a list of the cued operator schemas and the net weights of their cues. This list can be queried by the user at any point, so with `WeightListExecutor` it is unnecessary to register an `IOperatorCuedListener` (as was done in the `ListExecutor` example) unless fine-grain control is desired (such as to suspend the executor whenever an operator's cue exceeds a threshold, *e.g.* 1).

The `WeightListExecutor` manages the additive application of weights mapped to `IOperatorSchemas`, such that new schemas cued are added to the list, sorted by weight, and schemas receiving subsequent cues have their position in the list changed to reflect the combined weight. For weights δ in the range $(-1, 1)$, the rule for adding weights is simple addition; however weights such that $\lfloor |\delta| \rfloor \geq 1$ have a special transfinite behaviour, described later.

Evaluation Functions

Another application of the `WeightListExecutor` is to weight candidate operators in a manner similar to evaluation functions. As a simple example, we have implemented an alternative noughts-and-crosses strategy which runs a single feature detector, looking at each line and recording a feature `active-line` L n_{blank} n_{me} n_{opp} which counts the number of tokens belonging to the current player and the opponent. Each `active-line` then maps a cue to the operator for each blank square as follows:

- $n_{\text{blank}} = 1$ and $n_{\text{me}} = 2$: weight **2.0**
- $n_{\text{blank}} = 1$ and $n_{\text{opp}} = 2$: weight **1.0**
- $n_{\text{me}} > 0$ and $n_{\text{opp}} > 0$: weight **0**
- $n_{\text{blank}} = 2$ and $n_{\text{me}} = 1$: weight **0.3**
- $n_{\text{blank}} = 2$ and $n_{\text{opp}} = 1$: weight **0.2**
- $n_{\text{blank}} = 3$: weight **0.1**

This model is called `WeightListScoreRowsMain`, and is compared with the previous model, `WeightListMain`, in the next section.

Results

As before, we ran one million games with our models playing against a random player. `WeightListMain`, with the addition of the two features `near-opponent` and `in-wasted-line`, did significantly better than `ListMain`, losing only 0.07% of its games (down from 0.60%) and winning 90.96% (up from 90.41%; margin of error is 0.029%).

The evaluation-function player `WeightListScoreRowsMain`, with the single-feature weight cues, was even more aggressive, winning 90.91% of its games, but slightly less cautious, losing 0.20% (margin of error 0.026%). These statistics are shown in figure 3.5.

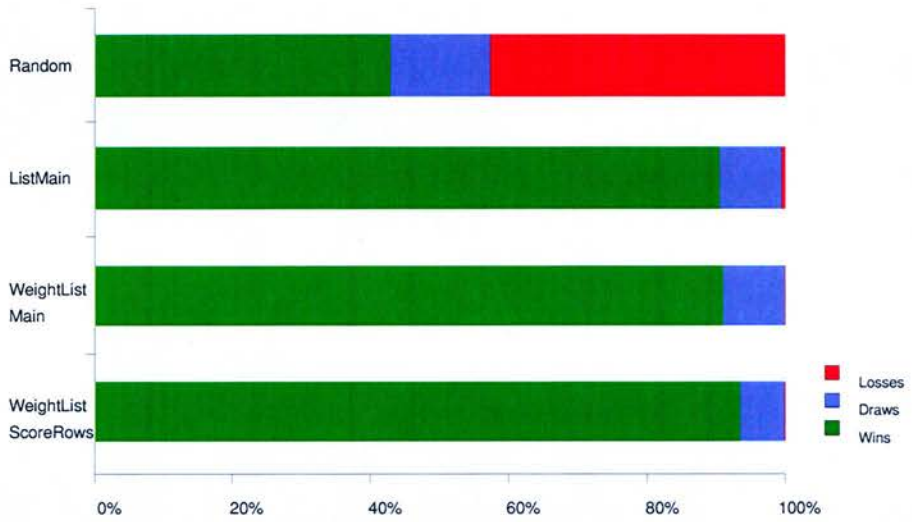

FIGURE 3.5: Performance of Weight List Players against Random

The ability to specify weights gives greater expressive power. Logically this is of course true, because `ListExecutor` could be implemented as a `WeightListExecutor` where the weight is constrained to be 1. Operationally, we have seen that in this domain a basic "feature list" strategy can be extended by using weights to overcome the problems observed. The `WeightList` approach can also be used to mimic evaluation functions, giving good performance with a small amont of code, using a single feature in the `ScoreRows` model.

Another point to make about this executor is that it allows detectors to run in parallel. Because the results are cumulative, it can be interrupted at any point and the best candidates applied in domains where there are timing limits. Efficiency can be improved by using "short-circuit evaluation", where an `IOperatorCuedListener` could stop the executor whenever an operator's weight exceeds 1.0. Alternatively, a "variable skill" player could be implemented by enforcing time limits, so say, the noughts-and-crosses player could have an "easy" level. Small random changes to the weights (noise) can be used to give stochastic behaviour, and a "variable skill' player could be made worse (an "easy" level in a game) by adding noise on the weights.

While these advantages might be useful in some contexts, the main focus of a problem solving system must be in performing well. The definition of good performance can vary, and for noughts-and-crosses we might desire either that a player win as many games as possible, or that the player lose as little as possible. `ScoreRows` has the highest win-percentage against the random player, but `WeightListMain` has the best no-lose record. It should be noted that all of the strategic players (`ListMain`, `WeightListMain`, and `ScoreRows`) have 100% draw records in contests with

themselves or other strategic players; thus, from the perspective of maximising the number of wins, the purely random player outperforms all the strategic players in head-to-head contests.

The ideal player would, if possible, have the highest win percentage *and* the lowest losing percentage against all possible opponents. To attempt to achieve this ideal, we will in the next section take a published optimal strategy for the game and see whether it can be implemented in the Feasch architecture.

Before trying this, let us look at some of the limitations of the `WeightList` execution strategy, with reference to these example implementations. Firstly, evaluating all the detectors is time intensive (the players in this section take about twice as long as the `ListMain` player); some processing could be eliminated by the "short-circuit" approach described above, but it would be preferable to constrain some of the detectors to run only when some features have been detected. (If we could do this, we could even leave all the noughts-and-crosses detectors in a problem-solving library used for many other purposes, such that they would only be activated if a feature such as `we-are-playing-noughts-and-crosses` is detected.)

Secondly, and related to the above limitation, natural and artificial strategies sometimes continue across several moves ("play at A and then on the next turn play at B") and make use of nested conditionals ("play at A, unless the opponent played at B or C, in which case run the specialised feature detector to check whether the opponent is trying to trap me, and if he is, play at D"). This was seen in Chapter 2. A mature execution strategy would permit the capture of this type of reasoning, which the `WeightListExecutor` does not.

Finally, the use of numeric weights leaves the model open to some arbitrariness. Where did our weights come from? What is to say someone else wouldn't choose different values, resulting in an incompatible model? This point will be revisited in later sections.

3.3.3. Networked Weighted Lists

What many of the limitations of the `WeightList` executor suggest is that it would be attractive to arrange the detectors in "tiers", whereby certain detectors are only run if other features are detected. The `NetworkWeightList` executor enables this, recording and combining weights on operator schemas (as `WeightListExecutor` does) and also permitting cues to other feature detectors. Both schema types are including them in the weight-ranked list of schemas cued at any point, with filtered views available to show detectors only (used internally to select which detector to execute) or operators only (available to the caller either to apply an operator or show the candidates to a user).

This executor combines all the capabilities of the previous two executors, and offers greater control of execution by specifying that detectors should only be attempted in the presence of certain features. Conceptually, it allows grouping detectors into "families" which are appropriate to certain problem sub-domains, and it allows for the expression of nested conditionals (by using weights to cue detectors; negated conditionals, such as the "unless" clause described previously,

can be implemented either directly, through inhibitory negative weights on detectors, or indirectly, through the use of explicit features for the absence of a specific feature). The `NetworkWeightList` executor also permits continuations (*e.g.* the prior example of "play at A and then on the next turn play at B") by allowing features to persist from turn to turn (so in this example a feature such as `just-played-at-A` could cue `play-at-B`; it might also cue exception checking, such as "unless the opponent played at B or C").

Since it appears to combine the various types of feature-based reasoning encountered in our survey, it is our recommended executor for problem solving with Feasch. The next section will describe the formal experimental conditions for testing Feasch, after which we will present our attempt to code a perfect-play noughts-and-crosses strategy in Feasch and then the results of this experiment.

Experiment: Does (PH) Hold for Noughts-and-Crosses?

To test the possibility hypothesis, that "features can be the basis of an AI problem solving system", we will take a published "expert" strategy for noughts-and-crosses and see whether it can be implemented in Feasch, using the most advanced executor `NetworkWeightList`. Our system can only differentiate between schemas by differing cues from features, so if we are able to encode a standard noughts-and-crosses strategy using multiple schemas or multiple features, we will demonstrate the possibility hypothesis. The hypothesis will *not* be demonstrated by putting everything into a single schema cued by a single default feature, and strong evidence (although not definitive proof) will be given for its falsity if we are *unable* to achieve this task. If we are successful, this in turn will provide motivation for more in-depth experiments exploring whether the feature-based approach is different to other approaches, and in particular whether it offers any unique benefits: showing that it is possible to perform AI problem solving with features is a far cry from showing that it might ever be desirable!

As has been noted, weaker `IExecutor` models have been shown to be insufficient for some types of problem solving; as it is reasonable to assume the same may be observed for `NetworkWeightList`, we can observe that the hypothesis is falsifiable, at least for the chosen strategy. We may also find that the strategy fits in the architecture, but very awkwardly: if we find that implementation is only possible using the trivial arrangement of a single default feature cueing the entire strategy, we will also conclude that the hypothesis is falsified — for this strategy — since the "feature" bears no relation to the domain. Although there is a possibility that the hypothesis (and architecture) may still have merit for other domains or other strategies, we will interpret such a failure, on such a basic problem solving task, as a significant disconfirmation of the hypothesis, and we will focus on its implications for the underlying cognitive theory.

There are many different strategies for perfect play in noughts-and-crosses. As the state space is small (on the order of 9! = 362880, less symmetries and early wins), search techniques such as A^* can solve the game in relatively small time. This approach does not generalise to more complex problems, however, and more succinct heuristic code can also give perfect play. Heuristic strategies have the added benefits of being easier to understand, debug, and remember. We will take the strategy for perfect play outlined and evaluated in Ostermiller (2005), described as follows:

- **Know the bad first moves.**

  Player 1. If you are going first, know the safe first moves. The trick is to avoid the edges. The corners and the center are safe moves.

  Player 2. There are two possibilities. Either player 1 took the corner, or the center. Safe moves for player 2's first move (player 1 in center): any corner. Safe moves for player 2's first move (player 1 in corner): centre.

- **Player 1 can be ruthless.** If player 1 moves in the corner for the first move, player 2 must take the center. If player 1 is playing against a novice, player 1 can be ruthless and always play in the corner first. That leaves a lot of board for novice to choose from and player 1 will win more often.

- **Become an expert.** The first moves (or opening book) are the hardest to figure out. Beyond the first move, it doesn't take much ...

The strategy described leaves some of the secondary details as an exercise to the reader. We completed the strategy as follows: The first player (X), on his second move, should check whether he can apply the "corner traps". If O took the centre (after X took a corner), X should play in the corner opposite his first move, and if X and O have both taken corners, X should take any remaining corner. The second player (O), on his second move, should check whether a corner trap is being applied, either X-O-X placed along a diagonal (play at any edge position, that is anywhere but a corner) or O in the centre with X's in a corner and at an edge-middle opposite (play at the corner nearest the tokens). Under any other circumstances, most naïve strategies can be used, including any of the ones described previously.

The "score-rows" technique is the naïve strategy we have chosen to use when the above strategy leaves play unspecified. This is because it is the most concise to specify, it is the most aggressive (in terms of win percentage), and because it raises the additional challenge of synthesising an evaluation function technique with the explicit conditional heuristic checking. Furthermore, "score-rows" makes the second corner trap check redundant, so for simplicity we will leave it out of our implementation.

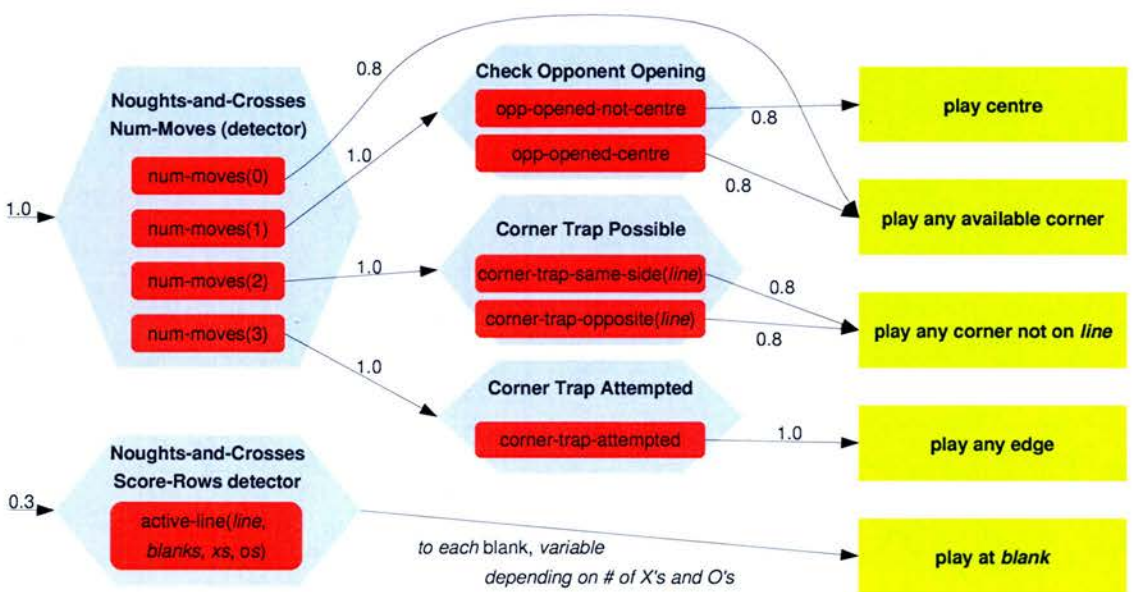

FIGURE 3.6: A Feature-Based Network Strategy for Noughts-and-Crosses

Graphically, the strategy can be visualised as in figure 3.6. In our implementation with `NetworkWeightListExecutor`, each of the blue hexagons represent feature detectors, testing for the presence of one or more features (indicated by red rounded boxes). The green boxes at right represent the operators cued by the various detectors. The leftmost arrows show the initial weight cues to feature detectors, and the other arrows show cues mapped to detectors or operator schemas when a feature is found.

Results

This `NetworkWeightList` strategy plays perfectly. It does not lose any games in any competitions with any of the strategies. Its percentage against the random player — 91.9% — is higher than any of the other strategies except `ScoreRows`, and in head-to-head competitions it defeats each of the other strategies. Against `ScoreRows` in particular, which plays very aggressively, it takes advantage of the recklessness which led to `ScoreRows`'s good result against the random player, defeating it in 39.8% of the games. (The other 60.2% of games are draws.) Against `WeightList`, which lost less than 0.1% of games against the random player, it wins 13.4% of the games (and again, the rest are draws). Full results are shown in figure 3.7.

		Player 2									
		Random		***List***		***WeightList***		***ScoreRows***		***Network***	
Player 1	***Random***	P1 Win	43.66%	P1 Win	0.60%	P1 Win	0.07%	P1 Win	0.20%	P1 Win	0.00%
		Draw	12.67%	Draw	8.99%	Draw	8.97%	Draw	6.40%	Draw	8.08%
		P2 Win	43.67%	P2 Win	90.41%	P2 Win	90.96%	P2 Win	93.40%	P2 Win	91.92%
	List	P1 Win	90.41%	P1 Win	0.00%	P1 Win	0.00%	P1 Win	0.00%	P1 Win	0.00%
		Draw	8.99%	Draw	100.00%	Draw	100.00%	Draw	100.00%	Draw	59.98%
		P2 Win	0.60%	P2 Win	0.00%	P2 Win	0.00%	P2 Win	0.00%	P2 Win	40.02%
	WeightList	P1 Win	90.96%	P1 Win	0.00%	P1 Win	0.00%	P1 Win	0.00%	P1 Win	0.00%
		Draw	8.97%	Draw	100.00%	Draw	100.00%	Draw	100.00%	Draw	86.65%
		P2 Win	0.07%	P2 Win	0.00%	P2 Win	0.00%	P2 Win	0.00%	P2 Win	13.35%
	ScoreRows	P1 Win	93.40%	P1 Win	0.00%	P1 Win	0.00%	P1 Win	0.00%	P1 Win	0.00%
		Draw	6.40%	Draw	100.00%	Draw	100.00%	Draw	100.00%	Draw	60.16%
		P2 Win	0.20%	P2 Win	0.00%	P2 Win	0.00%	P2 Win	0.00%	P2 Win	39.84%
	Network	P1 Win	91.92%	P1 Win	40.02%	P1 Win	13.35%	P1 Win	39.84%	P1 Win	0.00%
		Draw	8.08%	Draw	59.98%	Draw	86.65%	Draw	60.16%	Draw	100.00%
		P2 Win	0.00%	P2 Win	0.00%	P2 Win	0.00%	P2 Win	0.00%	P2 Win	0.00%

FIGURE 3.7: **Noughts-and-Crosses Contests.** The chart above shows the results of all pairwise contests between the five noughts-and-crosses player implementations presented here. The best performance is clearly shown by `NetworkWeightList`, which has no losses and outperforms the other players in nearly all metrics.

3.3.4. Discussion

As an AI problem solving system, the ability to write a perfect player for noughts-and-crosses is far from impressive — but an inability to do so would have been a nail in the coffin for Feasch. Subjectively, it was straightforward to develop the model, and, particularly with the `NetworkWeightList` executor, the model seems comparatively easy to understand and communicate. And although other methods can do the same task, *e.g.*, search (§2.1.2) and evaluation functions (*cf.* "score rows"), developing the solution in Feasch required what appears to be quite a different process. The diagram in 3.6 is not characteristic of these other techniques, and unlike usual search and evaluation function approaches, this diagram — and the Feasch implementation — corresponds very closely to a human-understandable strategy. Compare the code and diagram above with the following reference player, written in JavaScript with the same strategy (only the most relevant portions included; taken from Ostermiller, 2006):

```
function perfectMove(){
  var state = getState();
  var winner = detectWin(state);
  if (winner == 0){
    var moves = getLegalMoves(state);
    var hope = -999;
    var goodMoves = openingBook(state);
    if (goodMoves == 0){
      for (var i=0; i<9; i++){
        if ((moves & (1<<i)) != 0) {
          var value = moveValue(state, i, turn, turn, 15, 1);
          if (value > hope){
            hope = value;
            goodMoves = 0;
          }
          if (hope == value){
            goodMoves |= (1<<i);
          }
        }
      }
    }
    moveRandom(goodMoves);
  }
}
function moveValue(istate, move, moveFor, nextTurn, limit, depth){
  var state = stateMove(istate, move, nextTurn);
  var winner = detectWin(state);
  if ((winner & 0x300000) == 0x300000){
    return 0;
  } else if (winner != 0){
    if (moveFor == nextTurn) return 10 - depth;
    else return depth - 10;
  }
  var hope = 999;
  if (moveFor != nextTurn) hope = -999;
  if(depth == limit) return hope;
  var moves = getLegalMoves(state);
  for (var i=0; i<9; i++){
    if ((moves & (1<<i)) != 0) {
      var value = moveValue(state, i, moveFor, -nextTurn,
         10-Math.abs(hope), depth+1);
      if (Math.abs(value) != 999){
        if (moveFor == nextTurn && value < hope){
          hope = value;
        } else if (moveFor != nextTurn && value > hope){
          hope = value;
        }
      }
    }
  }
  return hope;
}
```

```
function openingBook(state){
  var mask = state & 0x2AAAA;
  if (mask == 0x00000) return 0x1FF;
  if (mask == 0x00200) return 0x145;
  if (mask == 0x00002 ||
    mask == 0x00020 ||
    mask == 0x02000 ||
    mask == 0x20000) return 0x010;
  if (mask == 0x00008) return 0x095;
  if (mask == 0x00080) return 0x071;
  if (mask == 0x00800) return 0x11C;
  if (mask == 0x08000) return 0x152;
  return 0;
}
```

Despite the apparent simplicity of the problem, this code for the solution is difficult to understand, and we suspect it was not easy to write[23]. "While human players understand the winning strategy very well, it is a challenge to translate this strategy into a computer algorithm" (Harris, 2001). It seems to us that one attraction of Feasch is that it facilitates expressing strategies which can be difficult or tedious to program in other systems. It seems that there is some amenability to modular design — because a complex strategy is implemented here by decomposing it into inspectable and reusable entities such as the features, feature detector schemas, operator schemas, and cues used — and there is a possibility of machine discovery for some aspects of the implementation, such as the numeric weights which we hand-coded in this instance. These points will be explored in subsequent chapters.

Our focus in this chapter has been to introduce the system and test the first, trivial but essential hypothesis, that structural features can be the basis of an AI problem solving system. The results conclusively confirm this possibility hypothesis; now let us turn to more interesting questions and to more challenging problem domains.

---

[23] The reader may be forgiven for suspecting that this code was deliberately obfuscated, or that we searched for a complicated solution. We offer our assurances that this is not the case: entering "tic-tac-toe ai program" into the Google search engine gives this as the third result, with the higher-ranking pages being a general article about game-playing and a question about 5x5 tic-tac-toe. Simpler solutions can be written, such as by using brute-force search, but a correspondence to understandable strategies is important to building scalable solvers and the implementation shown here, we reiterate, is the reference code for the strategy we used. Moreover, in our experience it is sadly indicative of how many AI systems are implemented.

Part Two: The Expressivity of Features

Chapter 4. Feasch and the Theorem Prover Isabelle

4.1. Motivation

Having seen in Chapter 3 that a system can be built to perform problem solving based on features, the next logical question to ask is whether this approach is actually different to other pre-existing techniques, and if so, how. To answer this question, we have chosen to apply the Feasch system to a well-developed area of AI problem solving where we can contrast it to a wide array of standard tools. After considering a number of potential areas, we chose to concentrate on formalised mathematical reasoning. This area of problem solving is extremely active, both in academia, with dedicated research groups at over 100 universities, and in industry, where it is an exciting and fast-growing sector. There are many varying approaches and systems for maths reasoning, which are used in a wide variety of ways, from fully interactive to wholly automated, and for a wide range of purposes. A number of standard techniques have been described for expressing control knowledge, and these serve as an important focus for comparing the current feature-based approach. Finally, if our approach does prove to be distinct in this area, it has the potential to be useful to a great many people, for theorem proving technology is used by a diverse range of communities, from research mathematics to financial security.

In this chapter, we will describe how we have integrated Feasch with a theorem proving system, first focussing on some of the existing challenges to semi-automated theorem proving, and then recounting the technical details of integrating the two systems. In the subsequent chapters, we will look in depth at how our approach using features differs from other approaches, specifically asking whether the inspectability of features is useful for proof assistants (Chapter 5), whether features have the same domain-specific utility as other techniques (Chapter 6), and whether they can perform well in a domain where other techniques languish (Chapter 7).

4.2. About Isabelle

Within the area of mathematical reasoning, we have chosen to work with the system Isabelle (§2.3; Nipkow *et al.*, 2006), because it is one of the most widely used in the area, with a good breadth of theories and a mature suite of end-user tools.

The Isabelle theorem prover is based on the HOL prover and so supports the usual higher-order logical expressions needed to formulate and reason about typical mathematical theorems. Theories have been developed in Isabelle for subjects ranging from complex analysis to Hoare

logic, and tools have been released into the public domain for authoring proofs in an Integrated Development Environment (Aspinall *et al.*, 2006), generating human-readable versions of proofs, and performing tactic-style planning (IsaPlanner, in Dixon, 2006). An active on-line community provides support for the Isabelle system and for most of these theories and tools. Many automation tactics are provided for solving and simplifying many types of expressions, and new tactics can be added to the system for easier proof development. As all proof steps are routed through a small verifier core which has been thoroughly examined by many researchers, the widespread confidence in the correctness of the Isabelle system extends across the range of theories and tools available.

4.2.1. Automation and Proof Assistants

Tactics in Isabelle

Currently, automatic theorem proving techniques in Isabelle are encoded as tactics written in ML. These tactics can perform powerful analyses of the specific domain for which they are written, and for many of the standard Isabelle library theories, particularly where decision procedures apply, very good tactics are supplied.

However, tactics have a number of limitations. Firstly, they are not easy to create, for to write a tactic, one must be familiar with the underlying ML code for Isabelle. This codebase is very large and very complicated, and it is not nearly as well documented as the more user-friendly end-user mode "Isar". Secondly, a tactic's behaviour is opaque; when it does not work, the output is usually just "proof command failed"; short of examining an enormous trace log, there is no way to observe what a tactic is doing. This makes tactics difficult to understand, more difficult to debug, and extremely arduous to adopt for a new domain. For these reasons, it is very difficult to extend tactics created by other developers, and as a result the Isabelle libraries consist of a small number of tactics which have been honed for particular tasks. Although these tasks include many commonly needed activities, there is a large area of potential automation for which no tactics exist.

The Simplifier

One particular family of tactics, simplifiers, are designed to be extensible, and the simplifier (*e.g.*, the tactic `simp`) is one of the single most powerful automation tools in Isabelle. Powerful simplification rules (corresponding to decision procedures) are included for common domains, reducing many expressions to canonical forms; frequently, this suffices to solve the problem, as "`1+1=2`" simplifies to "`true`". Developers can extend its behaviour by marking lemmas with "`[simp]`".

However, this technique often does more or less evaluation than is desired. If a user wants to inspect the proof found by `simp` for 1+1=2, she may be troubled to learn that there is no easy way to do this. If she wants to simplify only a part of a problem, or see why the simplifier

is doing something unexpected, she has to get deep into the innards of the Isabelle source code. Nor is this an idle worry: because simplification sets are designed with certain problems in mind, they can be impractical for other uses. Sometimes it can take an uncomfortably long time to run, and occasionally it can enter an endless loop and require interrupting or terminating the system. The simpset can be customised, but this is also a labourious process for which documentation is sparse. Even for experienced users, it can be difficult to spot conflicting rules. These conflicts often lead to over- or under- simplification or to infinite loops, and the uninspectable nature of Isabelle's simplification means these can be very hard to debug.

Because of these restrictions, most developers are circumspect about adding rules to the simplifier. The Isabelle manual advises that users should include only canonical simplifications, *i.e.*, only rules which are universally desirable, and while this is sensible in practice, it means that much useful control knowledge cannot be expressed as simplification rules.

IsaPlanner

A recent extension to Isabelle attempts to overcome the shortcomings of the use of tactics in general and the simplifier in particular. IsaPlanner (Dixon, 2006) provides a framework for writing proof plans (*v.a.* §2.3.iii, §4.2.2.ii) which resemble tactics but offer a number of advantages: IsaPlanner provides built-in contextual information and state caches to facilitate proof exploration (and to avoid endless loops); it allows introduction of explicit "gaps" so intermediate details can be postponed whilst following an overall plan; and, most usefully, its behaviour is easily inspectable and modifiable.

To date, IsaPlanner has principally focussed on automating inductive proofs, and plans implementing the rippling heuristic (Bundy *et al.*, 1993) have yielded excellent results. Although it is still in development, with documentation and the user interface still in relatively early stages, applications to other areas are already being explored. Its syntax is similar to that of Isabelle's tactics, and the increased functionality it offers makes it an attractive framework for guiding theorem proving, particularly in domains where success is not guaranteed. Whereas expressing qualitative and heuristic knowledge is not easily compatible with the approach of ML tactics (and particularly not with `simp`), IsaPlanner makes this convenient. However, like tactics, IsaPlanner takes a "top-down" approach, applying only those operators the developer explicitly included in the plan. Because these solution plans must be encoded holistically, it can be difficult to combine multiple plans or to reuse plan components. Also, as with tactics, there is no way for a developer to indicate when particular plans should be used (except for comments in source code or external documentation), and there is no way for the system to choose appropriate plans automatically.

A Possible Role for Feasch in Automation

This problem of retrieval is one where Feasch might be able to help. If knowledge components in Isabelle — theorems, definitions, tactics, and plans — were annotated with features, Feasch could try to detect these features in a problem state and then retrieve and apply potentially relevant tools automatically. The question we are asking of Feasch at this point is whether it differs from other approaches: if it is able to apply at this top-level, we will have some argument that it is somehow different to the other techniques.

The exploratory results of Chapter 3 also suggest other differences. None of the approaches described thus far provide reasoned explanations for their activity; in fact, only IsaPlanner gives much in the way of useful feedback beyond "Proof step failed" or printing the next proof state. We will argue in Chapter 5 that it would be difficult to use any of these techniques to generate such explanations, requiring extensions to the code for each step in the tactic or plan. If features are used to guide each step, we expect that this causal information will be immediately available; regardless of whether this information is useful (although we argue that it is), it highlights an important difference between the Feasch approach and the others. We have also noted that the modular re-use of control knowledge can be difficult with either tactics or plans; subsequent chapters will explore whether Feasch differs in this regard.

4.2.2. User Interfaces

Even with the most advanced tools, automated theorem proving falls far short of replicating the expertise of human mathematicians. As a result, much work in mechanised reasoning has focussed on the usability of systems for interactive theorem proving, where a human user guides the proof attempt (as described in §2.3.ii). The end product of this process is a body of formalised mathematics which has the correctness guarantees that machines can provide but which is much more sophisticated than current systems could produce automatically.

Unfortunately, this process can be extremely tedious, as mechanised proofs require enormous amounts of low-level detail which human mathematicians often find unnecessary[24]. For this reason, there is growing interest in semi-automated theorem proving, where a system incorporates automation techniques (such as Isabelle's `simp` tactic) within an interactive theorem proving context. These systems are commonly called proof assistants, and in many ways provide the best of both worlds: a user can observe an automated proof attempt, using its results when they are applicable, or instead manually guide the system when the automation flounders or the user has a good sense of the direction a proof should take.

[24] Consequently, such proofs are time-consuming both to write and to read; we will discuss this objection shortly.

Challenges to Semi-Automated Theorem Proving

Unfortunately, despite the apparent promise of the theory of semi-automated theorem proving, their adoption amongst research mathematicians has remained low. One primary reason given for this is given by Bundy (2006):

> After half a century, automated theorem provers are in a fairly mature state. A number of totally automated systems, such as Vampire, Spass and E, and a number of interactive systems, such as Isabelle, HOL, Coq, PVS, ACL2 and Nuprl, have a wide user base and a good track record in solving difficult problems. However, the use of the systems, even the totally automated ones, remains a highly skilled and time- consuming task. These act as a barrier to their wider adoption. In particular, and unlike computer algebra systems, automated theorem provers have never been popular amongst mathematicians.

In contrast to calculators and computer algebra systems, theorem proving systems cannot readily be used for a single task: the nature of formalisation requires that all the underlying mathematics be coded in the system before it can be used on a problem. As new theory libraries are being created all the time, this barrier is diminishing, but in its place, the problem of identifying relevant theorems grows with the number and size of libraries. One of the biggest and most time-consuming difficulties is being familiar with the rules and tactics available in a system.

Even when libraries are available and the user is familiar with their theorem names and other defined constructs, it is frequently the case that the amount of automation provided is so small that the system is unhelpful for exploratory mathematics. Computer algebra systems were adopted primarily because they automate and expedite tedious analyses, facilitating this exploration, and not because of strong reliability or accuracy guarantees. In fact, it is well known that computer algebra systems are unsound and error-prone, particularly at boundary situations, but they are useful enough, and right often enough, that they are a useful tool for mathematicians (particularly when exploring concepts which will be more thoroughly verified later if they lead anywhere). Theorem provers, on the other hand, are predicated on formal correctness, and the vast majority of tactics are based on decision procedures. When they are available, they work very well, but for more advanced (and arguably more interesting) areas of mathematics, there is no such guaranteed automation technique. (Where these decision procedure exist, furthermore, they are often available in more user-friendly computer algebra packages as well.) As a result, semi-automated theorem proving environments currently do not offer much assistance to mathematical exploration, as they are not usually geared towards approximate reasoning, with a few notable exceptions covered in the next section.

A more fundamental objection (Bundy, 2006) is that formal proofs are notoriously difficult for humans to understand, due to the enormous detail they require. Whilst some mathematicians relish this rigour in proofs (*e.g.*, Serre, from comments made at a meeting of the Royal Society), others are interested exclusively in the insights they afford (Cohen, same event). A consensus view is that proofs are "a vehicle for arriving at a deeper understanding of mathematical reality" (Aschbacher, same event), and the capability of theorem provers to drive this forward is limited. Most interactive systems, including Isabelle, support annotations for explaining proof steps and structuring them hierarchically, and there are a number of tools which can generate user-friendly versions of such annotated formal proofs. Automated tools, however, do not normally provide these annotations; in many cases, they *cannot*, because tactics often follow brute-force algorithms which, although suited to the nature of computer systems, do not follow human patterns of mathematical thinking and so fail to offer inspectability.

The Inspectability of Proof Planning

To be attractive to a wide audience of research mathematicians, semi-automated theorem proving environments must offer functionality for easily identifying relevant theorems and tactics, for representing approximate reasoning, and for usefully annotating the results of automated theorem proving. One approach which solves some of these issues is proof planning, introduced in §2.3.iii and implemented in IsaPlanner (Dixon, 2006)[25]. Closely related approaches have been used in other systems, *cf.* "proof levels" in Ωmega (Melis *et al.*, 1999) and "schemas" in Theorema (Buchberger *et al.*, 2006).

A proof plan, we recall, is an outline for a proof, usually following some pattern which has been identified as applicable to certain types of theorems. It differs from standard low-level tactics in two ways. Firstly, a plan will tend to run across many proof steps, using logic such as "after step 1, depending on factor *A*, apply step 2 or step 3 as many times as possible"; and secondly, proof plans will commonly modify a proof goal in a way which is not *a priori* logically justified, but which introduces new subgoals which have to be discharged later. Typically both the modified proof goal and the new subgoals are simpler than the original goal. It can be helpful to think of proof plans as capturing the human technique of reasoning "if we assume Z, then we can simplify the proof goal and then apply (some other technique), and if that's successful we can worry about proving Z later." Clearly such reasoning is not guaranteed to be useful, but for many types of problems, proof plans can capture general proof techniques at a greater level of abstraction than other formalisms. When good generalisations are found, the resulting proof plan is often extremely powerful, *e.g.*, solving induction problems with rippling (Bundy *et al.*, 1993).

Many strategies that mathematicians use can be easily mapped on to proof plans, and where this is done, the resulting proofs will follow standard, intuitive patterns. As proof plans are usually written in a hierarchical manner, these proofs, moreover, can usually be annotated in such a way that they can be expanded or collapsed to varying levels of detail. These two properties mean that proof plans can significantly help create machine proofs that are comprehensible, sometimes even attractive, to the human users. Furthermore, a user can observe a plan as it executes on a problem; if he detects a problem in the plan, say he sees that some step should be done differently, he could interrupt the plan, manually apply the alternate step, and then resume the plan.

---

[25] Eighteen months of this research was spent working with an earlier proof planning system, λClam (Richardson *et al.*, 1998), for which we developed a Java-based planning engine along the lines of Feasch. The underlying λProlog language it used was unreliable, and shortly after its developers stopped supporting it, the λClam project was cancelled. We ported our development to Isabelle, and although this took several more months, this new implementation benefits from Isabelle's advantages of usability and widespread usage. It is this Isabelle version of our work which is described in this thesis. IsaPlanner is emerging as the proof planning platform of choice for Isabelle, and we are in discussions with Dixon *et al.* about integrating it with Feasch. This is described more in §9.1.4.ii.

Semi-Automation and Isabelle

Isabelle has primarily been an interactive theorem proving environment, and its facilities for automation have historically been restricted to tactics implementing decision procedures. As discussed in §2.1.4, there are several limitations inherent in the tactic-based approach; with regards to semi-automated use, two of these limitations are especially noteworthy. Firstly, the all-or-nothing nature of tactics means that the only aspect a user can control is their initial selection; once invoked, there is no prospect of interacting with them. Secondly, there is no way in Isabelle to indicate when tactics should be used.

By combining a proof planning methodology with a tactics-like syntax, IsaPlanner overcomes the first of these limitations. The inspectability of proof planning — the fact that a user can observe what it the system is doing — lends itself to interactive usage, in addition to fully-automated proving. An IsaPlanner user can review each proof step in the plan (or each step that a comparable tactic would make). Wherever a plan has disjunctive branches, they are presented to the user as a list of possible next steps and he can select from among them. Alternatively, the user can specify that a plan should run automatically for a fixed period of time, after which he can review the entire space of proof exploration. At this point, he can select a state which seems promising and continue the proof attempt from there; he might let the system resume its automated exploration, he might manually choose among steps recommended by the plan, or he might specify some other proof step not in the plan. IsaPlanner plans do not currently have the breadth of coverage of the tactics in Isabelle, but the system is a very recent one. The development of new plans, for more types of problems in more mathematical domains, will likely increase the usefulness — and the appeal — of semi-automated theorem provers to mathematicians.

The second limitation, however, is not being adequately addressed by any system of which we are aware. When a user enters a new theorem to prove, Isabelle has no means of suggesting any automation technique or proof step. A few tactics, most notably `simp` (and closely related tactics, such as `auto`), are applicable very generally, and it is not uncommon for users of Isabelle to attempt "`apply simp`" blindly on any new state. Only afterwards, if the simplifier fails or does something undesirable, will these users analyse the problem and attempt to identify the appropriate tool, *e.g.*, a domain-specific tactic, possibly with a particular theorem, or an IsaPlanner plan. If the user can recognise what next step is appropriate — crucially, if the user is familiar with the relevant theorems, tactics, and plans — they can type it in to the system and proceed. Rarely, however, will the user be aware of all the relevant knowledge in the Isabelle library, and as this library is very large, finding it can be a challenge. Identifying relevant theorems can be complicated enough, but finding relevant tactics or plans is often even more difficult. Isabelle theorems (and definitions) are expressed in mathematical notation, and this is usually familiar, concise, and relatively easy to search; tactics and plans, however, have lengthy and complicated ML definitions, compounding the problem of identifying when they are relevant. Because it is so much easier to find individual

theorems, there is the danger that a user will ignore higher-level tactics or plans, and complete a proof at the lowest, most tedious level even when this process has been automated. Currently, the standard installations of Isabelle and IsaPlanner offer 80 tactics and 1 plan, so this problem is not so severe; for theorem proving to become widely useful, however, these numbers must grow, and as they do so, this danger will become very real.

In addition to potentially being useful for fully automated theorem proving, as suggested in §4.2.1.iv, Feasch may be able to assist in interactive proof by recommending plans, tactics, and theorems. If, when developing any of these knowledge entities, a developer specifies the features of a problem state which are relevant to its usage, then the system can automatically retrieve these entities when appropriate and present them to the user. This can be done for high-level tactics and plans as well as for the myriad individual lemmas and theorems in the library. By presenting candidate proof steps, Feasch has the potential of dramatically reducing the amount of searching and the amount of typing involved in writing Isabelle proofs.

4.2.3. The Eclipse ProofGeneral Environment

We suspect that if Feasch has significant differences to other control knowledge techniques, these differences will be related to the expressive style and inspectability of the approach. Central to any differences, we believe, will be the solution it offers to the problem of retrieval, suggesting proof steps on the basis of features. This could have ramifications for both automated and interactive theorem proving, and so it will be important to be able to evaluate Feasch with Isabelle in both areas. Other potential differences are that Feasch, by using explicitly defined features, might yield information about *why* a tactic is potentially appropriate, and that it could facilitate the re-use of control knowledge. As a semi-automated environment will be the best arena for exploring these prospects as well, let us survey what such environments are available for Isabelle. We will also look at how well Feasch could be integrated in this environment, both for our system to interact with the user and for it to interact with Isabelle.

About ProofGeneral

Far and away, the most widely used user interface for Isabelle is ProofGeneral (Aspinall, 2006). Initially developed as a plug-in for the Emacs editor, and continually supported and updated ever since, ProofGeneral offers a powerful, easy-to-use, text-based front-end for Isabelle (and, by using a generalised protocol, it can supports other provers as well). A typical theorem proving session with ProofGeneral involves typing proof commands in the editor and "sending" them to the theorem prover: if the command is applicable, it is "locked" in the display, the new proof state is displayed, and a user can enter another command (or "undo" the command(s) just sent and send different ones instead, *e.g.*, if the new proof state is technically valid but not an improvement to the previous one); if the command does not apply, the relevant error from the theorem prover is displayed and the user can edit or replace the command. At the end of a session,

the editor contains a file which records the final list of commands, providing new definitions, axioms, or theorems with proofs. The file can be saved, passed along to others, reopened, and re-executed in the theorem prover; if the file includes a proof to one or more theorems, it can be seen to constitute a reproducible formal verification of those theorems.

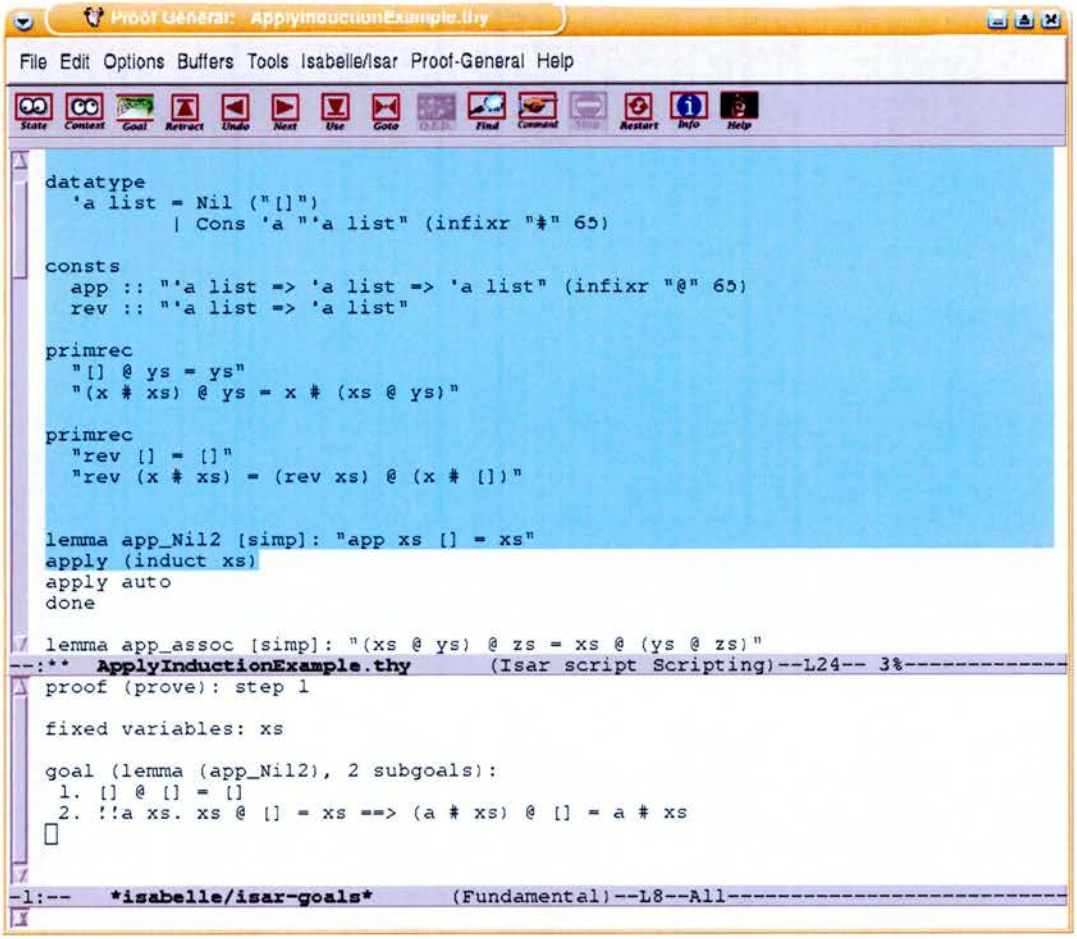

FIGURE 4.1: Emacs ProofGeneral

The automation components of ProofGeneral are exactly the automation components of the underlying theorem prover, *viz.*, Isabelle. The only "smart" assistance offered by the environment is a text-based search through certain libraries; any additional intelligent assistance (such as Feasch) would have to be written in Emacs Common Lisp, and developers have understandably preferred to work at the ML level.

To effect the underlying automation techniques, *e.g.*, `simp`, a user types them in like all other proof commands; in fact, there is no semantic difference in Isabelle between the use of complex automation techniques such as `apply simp` and simple rule applications such as `apply (rule` *theorem*`)`. IsaPlanner, likewise, is invoked by writing `apply (ipp` *technique*`)`. This begins a planning session in the Isabelle ML console in Emacs, where a user interacts with IsaPlanner via a command prompt; if the session is successful, the user can display the proof, cut it from the console window, and paste it into the prover file. As this all shows, Emacs ProofGeneral

is strongly text-based (unsurprising due to the nature of Emacs): it offers scarcely any of the graphical interaction features common in modern development environments.

To resolve this problem, Aspinall & Winterstein (personal communication) have been working on a new version of ProofGeneral based on the Eclipse platform, combining a full-featured source code editor with one of the richest set of graphical widgets. That project began to mature at the same time that we were looking to integrate Feasch with Isabelle in a user-friendly semi-automated environment; Eclipse ProofGeneral had the added attraction of being already implemented, in Java, with fully-functioning routines for communicating with Isabelle (through either process streams or socket streams), so it seemed an ideal starting point for the systems' integration.

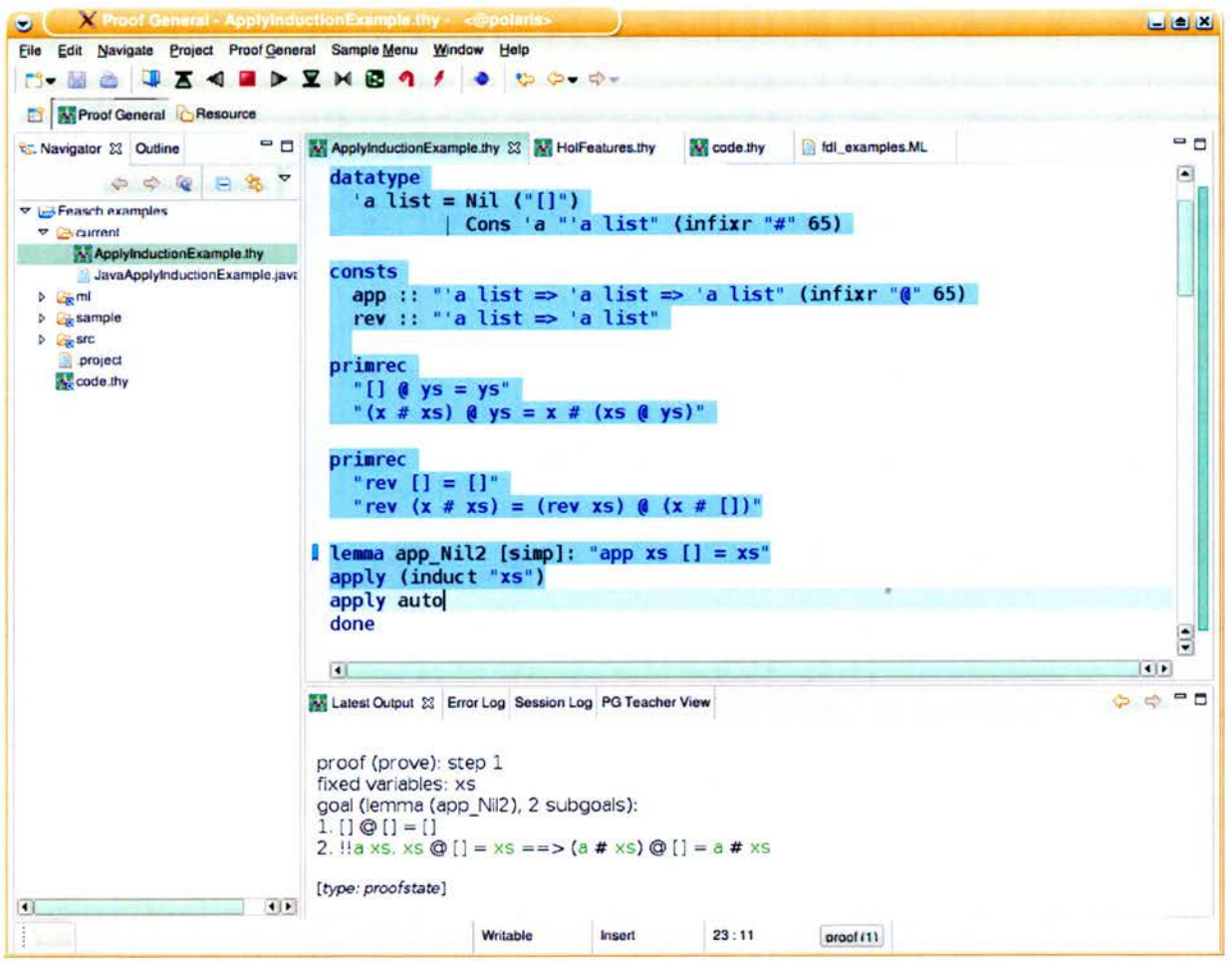

FIGURE 4.2: Eclipse ProofGeneral

Our Work on Eclipse ProofGeneral

The Eclipse ProofGeneral project ran out of funding soon after we chose to work with it, and as it was very nearly at the stage of being usable, we implemented a number of essential features and crucial bug fixes. Our aim in this effort was that if Feasch with Isabelle is to be

usable, all the underlying systems must be robust; and if the entire environment is attractive and easy-to-use, it will be easier to promote any benefits of Feasch with Isabelle that we find. This involved a significant amount of work, particularly as the Isabelle system was also undergoing major changes at the same time. (Our work to this end is summarised here for completeness's sake; this section may be skipped if the reader is interested in our system and not in the development we have done on Eclipse ProofGeneral.)

Many of the improvements we made had to do with the parsing and communications layer. We made a number of changes to the parsing routines so that Isabelle theory files can be handled reliably, and so that large files can be processed efficiently. Prover communications for parsing and sending commands were removed from the display thread (where they frequently hung the system) and routed through a new prover queue synchronised to resolve sequencing errors. Isabelle output was occasionally not valid XML, such as when special symbols such as < and & were used in proof goals, so we made the XML processing engine used by ProofGeneral more tolerant of these errors.

Another major area where we extended ProofGeneral was to add the ability to query and maintain Isabelle library information as Java data structures. Having a complete list of tactics and theorems is useful to the Feasch system — this is why we added the feature — but having this information also made it straightforward to provide context-sensitive on-line help and autocompletion within the proof script editor.

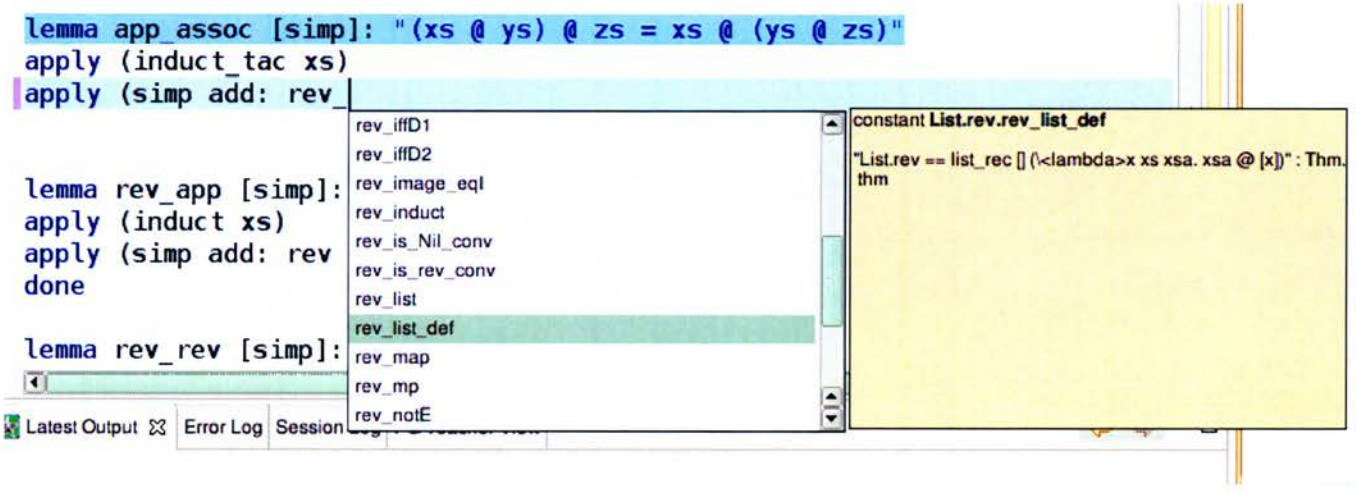

FIGURE 4.3: Autocompletion in Eclipse ProofGeneral

Finally we created a `ProverStandalone` interface to the ProofGeneral `SessionManager` which permits programmatic control of the theorem prover from Java without requiring the Eclipse editor or GUI. This is useful in ProofGeneral for unit testing, and also means Java programs can be written for controlling theorem provers, offering the Java-side benefits of high-level data structures, multi-threading, and internet support on top of the power and robustness of provers such as Isabelle.

All of these changes have been released to the public domain and added to the ProofGeneral codebase. They are independent of Feasch and it is expected they will be available in future versions of Eclipse ProofGeneral.

4.3. Integrating Feasch with Isabelle

By integrating Feasch with Isabelle, we are interested to discover whether the use of features and schemas offers much difference to the other techniques just described. Schemas, in our vocabulary, include IsaPlanner proof plans, complex Isabelle tactics, and low-level theorem applications, as well as feature detectors. Features, here, can be any attribute of a proof state that a developer judges as worth identifying; these might be superficial attributes, such as "the problem involves trigonometric functions", or "deep structure" aspects, such as "the conclusion of the proposition matches the conclusion of an implication in one of the assumptions". In the resulting system, a developer must be able to define the features, schemas, and mappings for her domain. She could do this by writing Java code, as we did for noughts-and-crosses (Chapter 3), with the Java code retrieving the proof state from Isabelle, running the detectors, and then calling to Isabelle to apply cued schemas (§4.3.1). From a usability perspective, however, it would be very much preferable to have access to Isabelle's rich semantics for proof state representation when defining and detecting features (§4.3.2), and to permit a developer to specify her Feasch control knowledge in the same file where she defines her theorems or tactics, using *e.g.*, a simple scripting language (§4.3.3). Finally, we will wish to have a user interface component, as part of the theorem proving session, where the schemas suggested by the control knowledge are presented to a user, in such a way that he can easily select one, have it sent to the prover and inserted in his proof, and then see the schemas suggested for the resulting proof state (§4.3.4).

4.3.1. Java

The first requirement for combining Feasch with Isabelle, as noted above, is to be able to query and modify the Isabelle proof state from Feasch Java objects; this is provided by ProofGeneral with our extensions. With this in place, we defined classes `IsabelleDomain` and `IsabelleContext`, fulfilling the Feasch `IProblemDomain` and `IProblemContext` interfaces, and providing prover-level and proof-state-level Isabelle functionality. The `IsabelleContext` constructor, as one important example, uses the `IsabelleDomain` and ProofGeneral routines to load a representation of the current goal in the Isabelle theorem prover. This representation is stored in an `IsabelleProblemState`; two other important classes are `GoalsHelper`, which assists in processing tree representations of Isabelle goals, and `IsarCommandSchema`, which stores commands for Isabelle proofs along with a textual description. Complete descriptions of these classes can be found in their JavaDocs.

Controlling Isabelle from Java

We will demonstrate how these classes can be used by developing a very simple feature-schema set for automating certain induction proofs, and invoking IsaPlanner's rippling technique if all else fails. The complete program is included on this project's web-page at `http://feasch.heneveld.org`, in the class `JavaApplyInductionExample`. We begin by using our ProofGeneral `ProverStandalone` class to control the theorem prover. The following code snippet initialises the theorem prover (with a precompiled theory with IsaPlanner and our code, described in the next section) and then waits for the prover to finish all start-up processing. The ellipsis will be filled in with code throughout this section. (The code following the ellipsis simply ensures that the session is closed and cleaned up after use.)

```
public void exampleListProof() {
  ProverStandalone prover = null;
  try {
    prover = ProverStandalone.createIsabelleWithLogic("HOL-IsaP-FeatWiz");
    prover.waitForAvailable();
    ...
  } finally {
    if (prover!=null) prover.dispose();
  }
}
```

At this point, we can send Isabelle theory file commands directly to the prover using the `sendCommand` method defined in `ProverStandalone`. The return value of this method, a `ProverCommandResponse` object, gives the result of the Isabelle command; it can be printed to standard output or queried to see whether errors occurred. In this example, we will wrap the call to `sendCommand` in a method called `showCommand` which displays the output from the prover. Filling in the body of `exampleListProof` we can write:

```
showCommand("theory ApplyInductionExample imports PreList IsaP begin");
showCommand("datatype 'a list = Nil (\"[]\") "+
   "| Cons 'a \"'a list\" (infixr \"#\" 65)");
showCommand("consts "+
   "app :: \"'a list => 'a list => 'a list\" (infixr \"@\" 65) "+
   "rev :: \"'a list => 'a list\"");
showCommand("primrec   \"[] @ ys = ys\" "+
   "\"(x # xs) @ ys = x # (xs @ ys)\"");
showCommand("primrec   \"rev [] = []\" "+
   "\"rev (x # xs) = (rev xs) @ (x # [])\"");
```

These commands set up basic definitions for a theory of lists; they are in fact taken directly from the Isabelle tutorial (Nipkow *et al.*, 2006), which then proves various simplification lemmas involving these constructs, leading up to the theorem that `rev rev x = x` for any list `x`. Let us send the first lemma definition to the prover, showing the corresponding output in a *`slanted font`*:

```
showCommand("lemma app_Nil2: \"app xs [] = xs\"");
      CMD> lemma app_Nil2 [simp]: "app xs [] = xs"
      proof (prove): step 0
      fixed variables: xs
      goal (lemma (app_Nil2), 1 subgoal):
       1. xs @ [] = xs
```

At this point we could continue manually sending the proof commands from the Isabelle tutorial:

```
showCommand("apply (induct xs)");
showCommand("apply auto");
showCommand("done");
```

Using Feasch for Programmatic Control of Isabelle

The proof above is not difficult, but one still has to be familiar with the tactics in order to use them. Feasch can be used to perform a simple feature-based analysis to identify these tactics when they might be appropriate. In this model, we will use two features. The first simply indicates whether "isabelle is active proving something" by checking whether the number of subgoals is greater than zero. It can be defined as follows:

```
public class ActiveIsabelleGoalDetector
extends BasicSchema implements IFeatureDetector {
  public final static String NAME = "isabelle is active proving something";
    public ActiveIsabelleGoalDetector() {
      super(NAME+" detector");
    }
    public void run(IProblemContext context) {
      if (((IsabelleContext)context).getProblemState().getNumSubgoals()>0)
        context.onFeatureFound(this, new SimpleFeature(NAME), null);
    }
  }
}
```

The second feature is more intricate, as it examines ProofGeneral's `ProverKnowledge` representation of the currently-loaded Isabelle libraries. This detector iterates through the elements in the conclusion of the goal, adding elements which look like inductively defined terms to one list and adding elements which look like variables to another. The feature "conclusion has inductive expressions" is added if the list of apparently inductively defined terms is non-empty. The detector also sets a mapping from the feature to an `IsarCommandSchema` specifying "`apply (induct var)`" for each of the variables *var* identified. The weight of the schema is computed as a function of the size of the conclusion and the frequency of inductive expressions.

```
public class InductiveExpressionDetector
extends BasicSchema implements IFeatureDetector {
  public final static String NAME = "conclusion has inductive expressions";
  public InductiveExpressionDetector() {
    super(NAME+" detector");
  }
  public void run(IProblemContext context) {
    int numInductives = 0, numTotal = 0;
    ArraySet<String> vars = new ArraySet<String>(),
       terms = new ArraySet<String>();
    for (Node n : GoalsHelper.getConclusion
       (((IsabelleContext)context).getProblemState().get())) {
      numTotal++;
      if (n.val.indexOf('.')>0) {
        String word = n.val.substring(n.val.lastIndexOf('.')+1);
        if (knowledge.getItem(n.val+"."+word+"_def")==null &&
            knowledge.getItem(n.val+"_def")==null &&
            knowledge.itemsStartingWithAndMatching(n.val+"."+word+"_",
               n.val+"\\."+word+"_"+".+"+"_def").size()>0) {
          //an entity xxx is recursively defined if it has definitions
          //(eg app_list_def) but not the usual xxx_def definition
          numInductives++;
          terms.add(word);
        }
      } else if (n.val.matches("[A-Za-z][A-Za-z0-9_]*") && !n.val.equals
         ("op") && knowledge.getItem(n.val+"_def")==null) {
        //note: even better would be to add only vars which are arguments
        //to the inductive terms identified above
        vars.add(n.val);
    }
    if (terms.isEmpty()) return;   //no inductively defined exprs
    IFeature f = new SimpleFeature(NAME, terms, 1.0d*numInductives / numTotal );
    MultiMapping map = new MultiMapping(true);
    for (String var : vars)
```

```
        map.add(new SchemaWithWeight<ISchema>(new IsarCommandSchema
           ("induct on "+var, "induct on "+var, "apply (induct "+var+")"),
            (numInductives+10.0)/(numTotal+10.0)));
      context.onFeatureFound(this, f, map);
    }
  }
```

With these two feature detectors defined, we can now set up the domain with appropriate mappings. For this example, let us send a cue of 0.7 to "`apply auto`" whenever Isabelle is active — since it's often a good thing to try — and add a small cue of 0.1 to "`pp ri`" (IsaPlanner with rippling) whenever an inductive expression is found. This small cue will normally be smaller than the variable cue sent by the detector above to "`apply (induct var)`", and the more computation-intensive IsaPlanner technique will be reserved for situations where other approaches fail.[26] The following code sets up the `IsabelleDomain` with the feature detector schemas and mappings just described. It can be called at any point once the prover is initialised; in our file, it is invoked just before sending the `app_Nil2` lemma to the prover (in the body of `exampleListProof`).

```
IsabelleDomain domain = new IsabelleDomain(prover);
domain.addSchema(new ActiveIsabelleGoalDetector());
domain.addSchema(new InductiveExpressionDetector());
domain.addMapping(new SimpleFeature(ActiveIsabelleGoalDetector.NAME),
   new SchemaWithWeight<ISchema>(new IsarCommandSchema("apply auto"), 0.7));
domain.addMapping(new SimpleFeature(InductiveExpressionDetector.NAME),
   new SchemaWithWeight<ISchema>(new IsarCommandSchema("pp ri"), 0.1));
NetworkWeightListExecutor ex = new NetworkWeightListExecutor(null);
```

We will now write the code for running the executor on a particular problem state. The following method repeatedly runs the executor, attempting to apply the cued operator schemas and recursing on the new state, until either a proof is completed or none of the cued schemas apply.

```
ProofRecord tryAutoProve(IsabelleDomain domain, NetworkWeightListExecutor ex)
throws InterruptedException, ScriptingException {
  IsabelleContext context = new IsabelleContext(domain);
  ProofRecord pf = new ProofRecord(context.getProblemState());
  ProverCommandResponse res = null;
  while (context.getProblemState().getNumSubgoals()>0) {
    ex.switchContext(context);
    ex.addDetectors(1.0, domain.getDetectors());
    System.out.println("RUNNING executor");
    ex.start();         //start running the executor (in bg thread)
    ex.waitForQuiet();  //then wait for it to finish
    ex.release();
    for (ISchemaWeightRecord sw : ex.getCuedOperators()) {
      if (sw.getSchema() instanceof IsarCommandSchema) {
        System.out.println("TRY suggested schema '"+sw+"'");
        res = showCommand(((IsarCommandSchema)sw.getSchema()).getCommand());
        if (res.fatalErrorEvent==null) {
          pf.steps.add(((IsarCommandSchema)sw.getSchema()).getCommand());
          break;
        }
      }
    }
  }
  if (context.getProblemState().getNumSubgoals()==0) {
    pf.completed = true;
    showCommand("done");
    pf.steps.add("done");
```

[26] Currently, IsaPlanner reports that proofs are successful even when they are incomplete, so that tactic really should be used with care! We include it in this example because it demonstrates the pattern of larger cues to those tactics which can be more easily tried, with some cue to the lesser used tactics so that they will not be overlooked, either by an automated prover or by a human user.

```
    pf.endMessage = "PROOF COMPLETED";
  } else {
    showCommand("sorry");
    pf.steps.add("sorry");
    pf.endMessage = "UNABLE TO COMPLETE PROOF";
  }
  return pf;
}
```

Now let us invoke the `tryAutoProve` tool. (As before, output in shown *slanted*. These commands follow the initialisation of the domain and executor in `exampleListProof`, replacing the commands for the manual proof of app_Nil2.)

```
showCommand("lemma app_Nil2 [simp]: \"app xs [] = xs\"");
     CMD> lemma app_Nil2: "app xs [] = xs"
     proof (prove): step 0
     fixed variables: xs
     goal (lemma (app_Nil2), 1 subgoal):
      1. xs @ [] = xs
ProofRecord pf = tryAutoProve(domain, ex);
     RUNNING executor
     TRY suggested schema 'apply auto[0.41176/1]'
     CMD> apply auto
     *** empty result sequence -- proof command failed
     *** At command "apply" (PGIP message).
     proof (prove): step 0
     fixed variables: xs
     goal (lemma (app_Nil2), 1 subgoal):
      1. xs @ [] = xs
     TRY suggested schema 'induct on xs[0.40741/1]'
     CMD> apply (induct xs)
     proof (prove): step 1
     fixed variables: xs
     goal (lemma (app_Nil2), 2 subgoals):
      1. [] @ [] = []
      2. !!a xs. xs @ [] = xs ==> (a # xs) @ [] = a # xs
     SUCCESS: suggested schema was applied; new goal is ((op = (...
     RUNNING executor
     TRY suggested schema 'apply auto[0.41176/1]'
     CMD> apply auto
     proof (prove): step 2
     fixed variables: xs
     goal (lemma (app_Nil2)):
     No subgoals!
     SUCCESS: suggested schema was applied; new goal is (No subgoals!)
     CMD> done
     lemma app_Nil2: ?xs @ [] = ?xs
pf.print();
     PROOF COMPLETED
     GOAL: ((op = ((ApplyInductionExample.app xs) ...
     PROOF:
        apply (induct xs)
        apply auto
        done
```

A complete proof is found, sent to Isabelle, and returned to the user. In fact, this simple Feasch model is able to solve all the simplification lemmas up to and including `rev_rev`:

```
lemma app_assoc [simp]: "(xs @ ys) @ zs = xs @ (ys @ zs)"
lemma rev_app [simp]: "rev(xs @ ys) = (rev ys) @ (rev xs)"
lemma rev_rev [simp]: "rev (rev x) = x"
```

With the addition of an analogous feature for applying `case_tac`, this can automate the proof of all lemmas in Chapter 2 of the Isabelle tutorial (Nipkow *et al.*, 2006). As the lemmas are carefully chosen and individually not difficult, this is not in and of itself an impressive feat. It is noteworthy, however, for two reasons. Firstly, it shows how control knowledge can be associated with Isabelle tactics so that they can be easily identified when they are relevant: instead of a user

having to search the libraries or the tutorial, Feasch can use this control knowledge to recommend methods automatically. Secondly, with Java controlling the theorem prover, other high-level automation techniques such as lazy thinking (Buchberger & Craciun, 2003) can be encoded in Java to help with both lemma speculation and automated proving.

4.3.2. Isabelle ML

Many of Java's capabilities may be useful for semi-automated theorem proving, including multi-threading, data structures, and regular expression testing for features. Some features, however, particularly those doing structural analysis of the proof state, could be done more easily detected using the Isabelle ML prover code directly. Furthermore, requiring features and schemas to be defined in Java is undesirable because this removes the control knowledge from the Isabelle files where tactics and rules are defined. For these reasons, we wanted to provide a mirror of many of Feasch's capabilities within the Isabelle ML session, culminating in an easy-to-use syntax where control knowledge can be specified alongside the relevant Isabelle definitions.

We have done this by defining analogous Feasch `Context` and `Domain` structures in ML, such that the ML `Domain` can store feature detectors and mappings, and the `Context` records the current ProofGeneral state (or an alternate state if provided).[27] Feature detectors can then be written in ML using Isabelle proof term internals; our Java code can then invoke these detectors by using ProofGeneral's Isabelle interface to execute "`ML {* FeaturesDomain.run_detector_on_context ... *}`". The ML domain includes methods for storing a "top-level" set of detectors and a set of mappings defined from ML; and provides methods for printing this information in an XML-like syntax which Java can parse. In this way, feature detectors and mappings can be defined in the Isabelle world and made visible to Java, so that the Java `Executor` can run them, optionally integrated with other Java-side schemas and mappings.

Detectors can use a rich library of ML search and unification algorithms when testing for features: the low-level term library in Isabelle offers routines for object-level analysis such as computing term size, and some more advanced facilities such as type-checking and eta-contraction. Yet more sophisticated functionality is provided by IsaPlanner, whose "focus term" partial unification algorithms have proved very useful. We have developed extensions to these routines which permit more flexible bi-directional unification and type instantiation, and we have put many tasks likely to be useful for a feature analysis in a uniform front-end package, `FSUnifyTools`. Of particular use are `is` and `has`, which look for entire and containment matches between two arguments, finding all schematic variable instantiations and — in one of the most challenging parts of the extensions — drawing type and sort inferences

[27] No analogous `Executor` is defined, as we will be driving the use of detectors from Java (ProofGeneral for semi-automated use or Java standalone), but such a structure could be defined to enable a pure-ML implementation of Feasch, *e.g.*, as a fully automated Isabelle tactic.

in both arguments. This is in contrast to the relevant library functions in IsaPlanner and Isabelle, where although schematic variables can be used in the two arguments simultaneously, certain matches may not be found, *e.g.*, if the types of these variables (possibly complex types corresponding to higher-order functors) are not specified on at least one side or iteratively updated. The problem is a difficult one, particularly if efficiency is important; because in our system, strict type-checking will necessarily be carried out later, we have made trade-offs in favour of speed and generality. The routine "`replace_vars_in_term (instantiations : (string*term) list) target`" in our library implements this, along similar lines to Isabelle's "`subst_Vars`", "`instantiate'`", and "`norm_term`", but tailored for more flexible use of schematic variables and types.

As with a lot of ML programs (in our experience), the details of the implementation are neither interesting nor easily explained. We supply extensive examples of low-level syntax, as well as commented code and documentation, with the source for Feasch with Isabelle in the folder `ml/`. We will not describe it in great detail in this thesis, preferring instead to insulate the user from these underlying complexities through a more user-friendly interface, described in the next section.

4.3.3. The Feature Description Language (FDL)

For Isabelle control knowledge to be easily represented through features and schemas, the developer should be able to define this knowledge in the same files where the relevant tactics and rules are declared, with most types of analyses expressible simply and clearly. One of the strengths of Isabelle is that formal mathematical knowledge is expressible with this simplicity and clarity: this has enabled the system to be widely used, understood, and extended. Providing the same support for the expression of control knowledge will, we expect, mean that heuristic applicability information can be shared along with the formal definitions.

We settled on a natural-language-style syntax with variable unification, where for example we can write "`the conclusion contains "rev ?X"`" to indicate that `rev` with one argument appears in the conclusion. In this example, `?X` is given the value of the argument; this might be ignored if it is irrelevant, or it can be set as a parameter in the feature. In addition to "`conclusion`", valid subjects in this grammar (left-hand-side words) are "`assumption`" and "`goal`", referring to the respective components of the current subgoal, and "`expression` *`Expr`*", referring to an arbitrary Isabelle term string *`Expr`*. (These can be preceded by an optional modifier, `some` or `the`, which has no effect but improves readability.) Valid verbs are `is`, for entire matches, and `contains`, for matches to a region of the source (`has` is provided as a synonym of `contains`). The "object" of the verb (the right-hand side) is expected to be a string which can be parsed as a term in the current Isabelle context, optionally using schematic variables such as `?X`. (The same is true for *`Expr`* on the left-hand side.) The detection engine finds all instantiations which satisfy the

statement (unless alternate behaviour is specified, perhaps for efficiency's sake), and the detector returns all the unique features that result.

For example, if we define a feature "`assumption_has_sum ["X","Y"]`" as "`(assumption contains "?X+?Y")`", when we compare it against a subgoal "`[ 1+z+y = x ; a+b = 1+z ] ==> x-y = a+b`", three features are detected:

```
assumption_has_sum (1, z)
assumption_has_sum (1+z, y)
assumption_has_sum (a, b)
```

Connectives `AND` and `OR` can be used to combine statements, and the modifier `NOT` can be used to negate statements. Unification sets are carried forward in order, with the usual semantics. Continuing the previous example, if we compared the feature definition "`((assumption contains "?X+?Y") AND (conclusion contains "?X+?Y"))`" against the same subgoal as before, only the instantiation `(?X=a, ?Y=b)` is returned; likewise, the definition "`((assumption contains "?X+?Y") AND (NOT (conclusion contains "?X+?Y")))`" returns just the other two instantiations.

There are three other statement forms recognised as feature definitions, useful for complicated connective statements. The first of these, `(instantiated "?X")`, returns true if *?X* has been instantiated (*?X*, of course, can be replaced by any other schematic variable, or even any expression; currently the implementation of `instantiated` detects whether the expression changes under application of unifications implied by earlier statements in the feature definition). The second form of statement, `(size "?X" TestFunction)`, computes the size of the instantiation of *?X* and applies it to a supplied ML *TestFunction* of type `int -> bool` (such as `(fn u => u<5)`, to check whether the size is less than 5; "size" is formally defined by the Isabelle system, but usually corresponds to the intuitive definition, where x has size 1 and `(x+y = Suc 0)` size 6). The third, `(atomic "?X")` returns true if *?X* is instantiated and its instantiation has size 1.

Instantiation checks are particularly useful to exclude degenerate unifications; if `"?P ?x ?y"` is matched against `0+1`, possibly unifications include such expressions as `(?P=λu v. u+1, ?x=0)`, `(?P=λu v. 0+1)`, and `(?P=λu v. v, ?y=0+1)` as well as the more obvious `(?P=λu v. u+v, ?x=0, ?y=1)`. By checking that all terms are instantiated, or checking that `?x` and `?y` are atomic, a developer can restrict features to particular desired scenarios. Note, however, that statements are evaluated in order, so if `(instantiated "?X")` occurs before the statement(s) specifying *?X*, it will never be satisfied.

One may also include arbitrary ML as a feature, as the `size` example suggests. The type of an arbitrary feature in ML is:

```
(string * Term.term) list  (* schematic variables already instantiated *)
-> (Term.term  (* the focus subgoal (usually the first) *)
    * Term.term  (* all goals *)
    * Feature.T list ref   (* the list of features *) )
-> ((string * Term.term) list (* a resulting set of consistent instantiations *)
       Seq.seq (* sequence of all possible such sets found *) )
```

This is substantially more complex than the FDL syntax we have described, but it permits a user to write more sophisticated detectors than we have allowed for, if and when required. It allows, for example, writing a detector which examines all subgoals or which looks at the features found thus far in the analysis.

Within an Isabelle theory file, features are defined by writing ML code "`feature` *`name vars statement`*`;`", where *`name`* is a string specifying a feature name; *`vars`* is a list of strings specifying the schematic variables that should be included as part of the feature (without the `?` prefix); and *`statement`* is a feature definition as described above (including the possibility of arbitrary ML code). Using the Feature Description Language and this command, we can now define features similar to those in the previous section for a theory about lists. Our theory file starts out as before:

```
theory ListFeaturesExample imports PreList IsaP
begin
datatype
  'a list = Nil ("[]")
          | Cons 'a "'a list" (infixr "#" 65)
consts
  app :: "'a list => 'a list => 'a list" (infixr "@" 65)
  rev :: "'a list => 'a list"
primrec
  "[] @ ys = ys"
  "(x # xs) @ ys = x # (xs @ ys)"
primrec
  "rev [] = []"
  "rev (x # xs) = (rev xs) @ (x # [])"
```

We then define two features, one which is detected whenever Isabelle has an active goal and another which detects occurrences of `rev` and `app` in the conclusion (possibly returning multiple features).

```
ML {*
feature "isabelle is active" [] (the goal is "?X");
feature "the conclusion uses recursive list expressions" ["X"]
  (((the conclusion contains "rev ?X")
    OR (the conclusion contains "app ?X ?Y")
    OR (the conclusion contains "?Y @ ?X"))
   AND (size "?X" (fn u => u=1)));
*}
```

The second feature definition includes arguments to `rev` or `app`, provided they are of size one. The reason for restricting it to expressions of size one is that we are interested in identifying variables over which induction could be performed. The test we perform is neither necessary nor sufficient (`?X` could be instantiated to `Nil`, for example, but not to `A` in the expression `rev A @ []`) but rather it is a simple approximation of appropriateness. In contrast to Isabelle theory items or most tactics, features and mappings permit rapid formulation of very rough control knowledge. A Feasch model can, of course, be refined — by the developer or by another user — but even in a simplistic form, and without much work on the part of the developer, (such as our "`uses recursive list`" feature), the approach seems to provide information about a domain that is different to other approaches and potentially quite powerful. These themes of expressiveness and approximate reasoning, as well as modularity and re-usability, will be explored in subsequent chapters.

The second definition above demonstrates also that Isabelle term expressions can be written with valid Isabelle shortcuts; `"?Y @ ?X"` is equivalent to `"app ?Y ?X"`. The detector returns all unique features, so for the expression `"A @ B = rev ((rev B) @ (rev A))"`, it will find two features, "`the conclusion uses recursive list expressions (A)`" and "`the conclusion uses recursive list expressions (B)`".

In addition to defining features within Isabelle theory files, it is desirable to be able to specify which detectors should be run initially and what should be cued when features are found. The command "`mapping` *`name vars cue-list`*;" specifies that the feature *`name`* should cue the schemas in *`cue-list`*; the list *`vars`* consists of strings specifying variables which are assigned the values of arguments in the feature. Whenever these variables appear as schematic variables (*i.e.,* prefixed by `?`) in *`cue-list`* schemas, they will be replaced by the corresponding feature argument. The *`cue-list`* is a list of (*`schema`*, *`weight`*) pairs, where *`weight`* is a real number, and *`schema`* is of the form (`ProofCommand` *`command`*), where *`command`* is a string representing an Isabelle command (such as `"apply auto"`); (`NamedProofCommand` (*`name`*, *`command`*)), where *`name`* is a textual description of the command (such as `"Use the 'auto' tactic "`); (`FeatureDetector` *`detector`*), where *`detector`* is a string corresponding to a feature name; or (`NamedSchema` *`name`*), where *`name`* is a known schema identifier (defined either in ML or in Java). With these commands, we can define our mappings as follows:

```
ML {*
mapping "isabelle is active" []
  [ (ProofCommand "apply auto", 0.7),
    (FeatureDetector "the conclusion uses recursive list expressions", 1.0) ];
mapping "the conclusion uses recursive list expressions" ["X"]
  [ (ProofCommand "apply (induct ?X)", 0.5) ];
*}
```

Finally, a user can specify that a feature detector should be run initially with the command `add_top_level_schema` *`schema`*. (It is also possible to specify multiple sets of top-level detectors with the `FeatureTool` set of commands; for this, and for a lower-level description of the other commands described here, see `fdl_examples.ML` in `ml/examples`.) Our Feasch model in Isabelle, comparable to that described in Java in the previous section, is complete when we include the command:

```
ML {*
add_top_level_schema (FeatureDetector "isabelle is active");
*}
```

When ProofGeneral (and applications using `ProverStandalone`) have parsed these commands, it has access to these `FeatureDetectors` and `ProofCommand` schemas. An executor using the top-level schemas in the Java `IsabelleDomain` will, in this case, start with the "`isabelle is active`" detector, and receive the cues for the command "`apply auto`", the "`recursive list`" detector, and "`apply (induct ?X)`" if appropriate (with *`?X`* instantiated as per the recursive list features). The method `tryAutoProve` would work, exactly as before, trying to prove various lemmas using the methods cued at each proof step. Unlike in the previous section, however, the control knowledge does not need to be described in Java; a developer can specify it at the same time as she defines her theory.

4.3.4. Tying it All Together: The Feature Wizard

For the Feasch model to be widely useful, its execution and suggestions should be available in the same semi-automated theorem proving environment where a user would ordinarily work. This means it should be accessible in or alongside the ProofGeneral editor. To this end, we developed a front-end which sits as a "view" in Eclipse, called the "Feature Wizard" and implemented in `org.heneveld.featwizpg.FeatWizMainView`.

The Feature Wizard monitors the current ProofGeneral Eclipse proof script and proof state, and whenever an unproven subgoal is active, it can invoke a Feasch `Executor` using detectors, mappings, and top-level schema sets from Isabelle or Java. `FeatWizMainView` includes routines similar to those presented in §4.3.1 together with graphical controls and status indicators. The features detected and the methods (operator schemas) cued are listed in two tab panes as shown in figure 4.4.

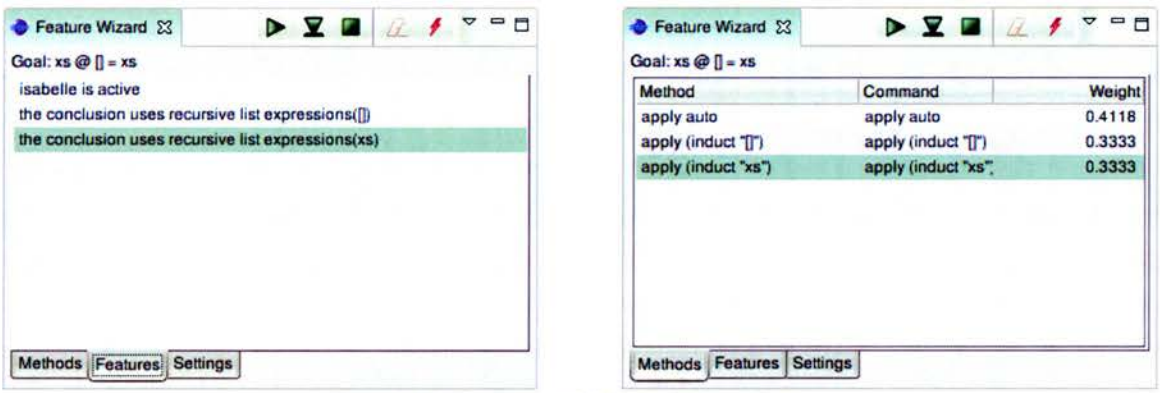

FIGURE 4.4: Feature- and Method- List Tabs

This view can be docked in Eclipse and visible alongside the proof script editor and other views (such as the proof state view) as shown in figure 4.2. By clicking on a method, a user can have it inserted in the proof script editor and sent to the prover. If it can be successfully applied, the prover remains in the new state, the new proof goal is displayed, and the user has the option of entering further command manually or running the Feature Wizard on the new goal state. If a cued command is selected and inserted but the prover determines it is not actually applicable (which can happen when mappings specify likely commands without computing logical necessity, similar to a user who will often try "`apply auto`" just in case it works), the Feature Wizard removes it from the editor and places a red "error" icon next to the method in the list; a user could then try another candidate method from the list or manually enter a command (as in normal ProofGeneral usage).

The Feature Wizard is designed to be a "minimal impact" GUI. It can be present and operational without requiring a user to change her working habits at all; if a user sees useful suggestions appearing, however, she can—and we hope will—incorporate the Wizard into her way of working. The behaviour of the Wizard can be customised in another "Settings" tab, allowing control of such items as whether the Feature Wizard runs automatically and whether high-scoring methods should be automatically tested and, if successful, inserted in the proof script. These items can all also be overridden by manual clicks on toolbar commands to "run" the executor, "stop" it, or enter fully "auto"matic mode; these commands, and experiments involving Feasch and the Feature Wizard, are described in the next chapters.

Chapter 5. Interactive Theorem Proving with the Feature Wizard

We will begin our exploration of the "Feasch with Isabelle" system by looking at some of the most tedious steps in theorem proving. Nearly all proofs comprise a significant portion of common applications of elementary logical rules: typing these in can be one of the most laborious parts of using Isabelle. By defining features to suggest when these rules apply, the system could propose them automatically, saving the user the time of typing, as well as the risk of typing errors. Because it is frequently necessary to type lengthy instantiations into the applied rules, the advantages of such a tool could be considerable.

5.1. Globally Useful Features for Isabelle/HOL

Let us start with one of the most trivial parts of the proof. Very often, the last step before completing a goal is to tell the prover to "`apply assumption`"; while this is not hugely taxing for the user, it is a reasonable expectation that a proof assistant do this automatically when possible. We can do this with Feasch by defining a feature that "an assumption matches the conclusion", mapped to the proof step. Because most proofs consist of many subgoals, "`apply assumption`" is one of the most common steps; thus it will be useful to have it easily available for insertion. Even if it is not difficult mathematically to identify when to use it, this is, to our knowledge, the first time it is available interactively for Isabelle. The FDL code (§4.3.3) for this is:

```
feature "an assumption matches the conclusion" []
    ((the conclusion is "?P") AND (some assumption is "?P"));
mapping "an assumption matches the conclusion" []
    [ (ProofCommand "apply assumption", 2.0) ];
add_top_level_schema (FeatureDetector "an assumption matches the conclusion");
```

The "detector schema" for this feature uses the conclusion to set a schematic meta-variable and then checks whether any assumption also matches that variable. As mentioned in §4.3.2, we have developed specialised ML functions for this unification; one consequence of this is that `?P` could unify with an assumption and a conclusion which are not literal matches, as long as they are unifiable through eta-contraction and expansion or by instantiating other schematic or unbound variables (*i.e.*, `?P` will match a conclusion "`∃u. A u`" with any of the expressions "`∃v. (λx. A x) v`", "`Ex A`", or "`∃v. ?H v`" as assumptions).

5.1.1. Cueing the Obvious

In addition to "apply assumption", there are a number of other rule applications which are often desirable whenever they are valid. Just as we tested for "an assumption matches the conclusion", simple features are one way to record when these rules are appropriate. Looking at sample proofs, we identified four such rule types which occur with high frequency. These deal with conjunctions (conjI/E), implications (impI/E), universal quantifications (allI/E), existentials (exI/E). We note that there are many other techniques which apply some of these normalisation strategies, but for the moment we will forget about those techniques and focus on what — and how — control knowledge can be expressed using Feasch. There will be ample comparison with other techniques towards the end of this chapter and in subsequent chapters.

For each rule type above, there are instances where the destruction of the logical operator is a safe and almost universally desirable thing to do. When an assumption is a conjunction of two propositions, it is nearly always easier to remove the conjunction and make each proposition an explicit assumption. Likewise, if there is an implication or a universal quantification in the conclusion, or if an assumption is an existential, it is usually good to remove the relevant operator. It is easy to identify the features to detect each of these situations, and with Feasch, they can be encoded as follows:

```
feature "some assumption is a conjunction" ["P","Q","PQ"]
    ((some assumption is "?P & ?Q") AND (expression "?P & ?Q" is "?PQ"));
feature "the conclusion is an implication" ["P","Q"]
    (the conclusion is "?P --> ?Q");
feature "the conclusion is a forall quantification" []
    (the conclusion is "ALL x. ?Q x");
feature "some assumption is an existential" ["P"]
    (some assumption is "EX x. ?P x");
```

It is also necessary to add mappings from these features to the relevant Isabelle commands, invoking the destruction-rule or elimination-rule tactic for the logical rule. User-friendly names for these schemas are specified, for although this is not necessary—and was not done for the simple "apply assumption" schemas—we feel that it may increase readability and inspectability. Using "expression" to bind additional schematic variables, as done with ?PQ above, can be useful to this end. These names are not an integral part of the system, except as labels for the mappings, and they can easily be suppressed in the graphical user interface (GUI) in favour of viewing just the commands. Alternatively, a user could edit the control knowledge to use a more terse or otherwise preferred format. However, for libraries where a user is not familiar with all the theorems, and for novices moving to Isabelle, we suspect that a natural language description will prove useful. and we will investigate one possible use in §5.2.

```
mapping "some assumption is a conjunction" ["P","Q","PQ"]
    [ (NamedProofCommand ("Separate conjunction assumption ?PQ",
          "apply (erule_tac P=?P and Q=?Q in conjE)"), 1.1) ];
mapping "the conclusion is an implication" ["P","Q"]
    [ (NamedProofCommand ("Move ?P from conclusion implication to assumptions",
          "apply (rule impI)"), 1.2) ];
```

```
mapping "the conclusion is a forall quantification" []
    [ (NamedProofCommand ("Remove forall quantification from conclusion",
         "apply (rule allI)"), 1.2) ];
mapping "some assumption is an existential" ["P"]
    [ (NamedProofCommand ("Instantiate assumption existential ?P.",
         "apply (erule_tac P=?P in exE)"), 1.3) ];
```

One thing to note is that some command schemas specify instantiations (using tactics where `_tac` has been appended), but that this is not necessary here for the rules applied to the conclusion (using the low-level tactic `rule`, as opposed to the other low-level Isabelle tactics for rule application, `erule`, `frule`, and `drule`, which apply for elimination, in a forwards direction, and in a forwards direction destroying a used assumption, respectively). The application of these rules to the conclusion is guaranteed to be uniquely defined, so the simple form can always be used here. This is not always the case, and where these rules are applied to assumptions, the simple form might be ambiguous. Isabelle will apply these rules to the first matching assumption, but specifying the instantiations offers several advantages. This gives the user control over the rule applied, it makes the resulting proof robust if the order of assumptions is changed, and it means the rule application can occur more quickly.

A second point to note is that while the mappings above ensure that the associated schemas will be cued when the detectors are run, we have not told Isabelle when to run any of these detectors. Because these feature detectors are simple, we will specify them as top-level schemas, meaning that they will run without being cued.

```
add_top_level_schema (FeatureDetector "some assumption is a conjunction");
add_top_level_schema (FeatureDetector "the conclusion is an implication");
add_top_level_schema (FeatureDetector "the conclusion is a forall
    quantification");
add_top_level_schema (FeatureDetector "some assumption is an existential");
```

Under the hood, all that this command does is indicate that these detectors receive an initial cue of 1.0. This value can be adjusted — this is described in comments in the ML code — but an easier way to encode more complex behaviour is to use the `NetworkWeightList` executor to chain schemas. One detector (such as "`isabelle is active`" from §4.3.3) can be used as the initial trigger, and mappings added from this feature to other detector schemas, so that they are only run when the relevant feature is detected. The chains can even be longer: for instance, the top-level schema could cue a detector which looks for an existential anywhere in the goal, and if this feature is found, the "`some assumption is an existential`" detector could be cued. This is most useful when a detector may be computationally expensive and necessary only in specific instances (so not for these examples, although we will shortly introduce cases where this is useful). Note that one feature detector must always be specified as a top-level schema, and others triggerable from this via some sequence of detectors (or possibly other schemas, such as tools), otherwise the detectors will never be invoked by the system.

5.1.2. Heuristic Recommendations

These rules are frequently used in other ways where desirability is more difficult to detect. Using the network executor to chain schemas, and using numeric weights on mappings, we can encode this heuristic control knowledge in Feasch. Separating conjunctions in the conclusions, for example, is often done, but is not always necessary, and is typically delayed until after assumptions are massaged into desired forms (so as not to duplicate this work when proving each conjunct). As a consequence, we will use a smaller weight — 0.5 — for the mapping between the features and schemas for this command.

```
feature "the conclusion is a conjunction" [] (the conclusion is "?P & ?Q");
mapping "the conclusion is a conjunction" []
    [ (NamedProofCommand ("Separate conjuncts in conclusion
          (to prove them separately)", "apply (rule conjI)"), 0.5) ];
add_top_level_schema (FeatureDetector "the conclusion is a conjunction");
```

In general, we use the convention that a weight $\omega \geq 1.0$ should be used for schemas which are universally desirable, such as those in the previous section, and $\omega \geq 2.0$ should be restricted to those universally desirable commands which discharge a subgoal. By sending a weight of 0.5 to the command "`separate conjuncts in conclusion`", less than those used in the previous sections, we ensure that this command will be ranked lower in the list. Because this command will normally be done only when the others do not apply, this is appropriate. If, for example, we have "$A \wedge B$" both in the assumptions and as our conclusion, we should prefer to "`apply assumption`" ($\omega = 2.0$) rather than separate the conjunction in either the conclusion ($\omega = 0.5$) or the assumption ($\omega = 1.1$).

As described in §3.3.2, a $\mathbb{Z}$-transfinite sigmoidal function is used for successive cues. What this means in practice is that two cues of 1.1 will result in a combined cue which is *less* than a single cue of 2.0. A cue of 1.0 followed by 0.1 gives the same net result as a single cue of 1.1, which is greater than a single cue of of 1.0. Three cues of 0.5 is equal to two cues of 0.75, and both are less than a single cue of 1.0. This may be hard to get used to initially, but our hope is that users need not concern themselves overly much with finding the exact weight values. We hope that good behaviour will result from following the convention above, using rough estimates for the fractional part of each weight. Relying too much on precise fractional values could result in a system which is too inflexible and inextensible — these problems and some possible solutions are discussed in §9.1.3— but for now let us note that we recognise it as one of the most *ad hoc* (or "below the line") aspects of our system. However, as they are similar to devices used elsewhere (*e.g.*, evaluation functions from §2.1.4 and fuzzy logic from §2.2), we expect that such weights will be a relatively easy way to get powerful behaviour, and one with which users will be familiar.

5.1.3. Hierarchical Executive Control

As mentioned at the end of §5.1.1, the `NetworkWeightList` executor can be used to structure execution hierarchically. In subsequent chapters, for example, we will look at using features to solve derivative and integral problems; if there is no derivative or integral in the problem, clearly it would be a waste of resources to run any of these detectors. Using mappings to set up a hierarchical network of schemas, we can ensure that some detectors — particularly those which may use a lot of resources — are run only if we have found features suggesting they are worth doing. Furthermore, the weights in the mappings influence the order in which cued detectors are run. Our expectation — which will be explored in due course — is that this supports and encourages the development of complex feature detectors, without impairing performance.

The last section showed how separating the conjuncts in a conclusion can be suggested as a weaker recommendation than other rule applications. They are times, however, when it is actually preferable to these other schemas, *viz.*, when one of the conjuncts can be easily proved. It would be grossly inefficient to do this type of analysis when the conclusion is not a conjunction, but if we have already found the feature "`the conclusion is a conjunction`", it would be useful to see whether either of the conjuncts can be easily proved by assumption. We encode this judicious execution of a more complicated feature detector by defining a mapping from the simpler feature:

```
mapping "the conclusion is a conjunction" ["P","Q","PQ"]
    [ (FeatureDetector "the left-hand conjunct of the conclusion matches an
        assumption", 0.8),
      (FeatureDetector "the right-hand conjunct of the conclusion matches an
        assumption", 0.8) ];
```

We now have a hierarchical network of detectors, and the features and mappings can be defined in a similar way as before:

```
feature "the left-hand conjunct of the
    conclusion matches an assumption" ["P","Q"]
    ((the conclusion is "?P & ?Q") AND (some assumption is "?P"));

mapping "the left-hand conjunct of the
    conclusion matches an assumption" ["P","Q"]
    [ (NamedProofCommand ("Discharge conclusion left-hand conjunct ?P by
        assumption", "apply (rule conjI, assumption)"), 2.0) ];

feature "the right-hand conjunct of the
    conclusion matches an assumption" ["P","Q"]
    ((the conclusion is "?P & ?Q") AND (some assumption is "?Q"));

mapping "the right-hand conjunct of the
    conclusion matches an assumption" ["P","Q"]
    [ (NamedProofCommand ("Discharge conclusion right-hand conjunct ?Q by
        assumption", "apply (subst conj_commute, rule conjI, assumption)"),
    2.0) ];
```

Both features are needed because "`subst conj_commute`" is necessary when the it is the right-hand conjunct being discharged. The weights of these mappings are set to 2.0 because these commands discharge goals.

An Example

As an example, consider "`lemma "[| a & c ; b |] ==> a & b"`" with the model described so far. When the user sends this lemma to Isabelle, it is parsed, stored internally as the "current proof state", and returned in a parsed form to Eclipse ProofGeneral. ProofGeneral then passes the returned expression to various display components, which render it for the user to view, and to Feasch, which begins the feature analysis. Feasch invokes the first top level schema, the detector "`an assumption matches the conclusion`", sending it to Isabelle for evaluation and receiving the reply that no feature is found. Feasch then tries the next top level schema, the detector "`some assumption is a conjunction`", and finds that the feature "`some assumption is a conjunction (a,c)`" is present. This is added in the "Features" view of the Feature Wizard, and because it is mapped to the command schema "`Separate conjunction assumption "a ∧ c"`", this command is added in the "Methods" view of the Wizard. Feasch continues executing the detectors, according to their weights (following the order they were added if the weights are the same, *e.g.*, for top-level detectors). When it finds the feature "`the conclusion is a conjunction`", Feasch adds the feature to the "Features" view, it adds the command "`Separate conjuncts in conclusion`" to the "Methods" view, and it cues the second-tier detectors for checking the conclusion's left- and right-hand conjuncts against assumptions (with a weight of 0.8 specified in the previous section, so these detectors will run after the other top-level detectors). The detector for the right-hand conjunct finds that feature and recommends the relevant command for proving that part of the conclusion. The final view of the methods is shown in figure 5.1.

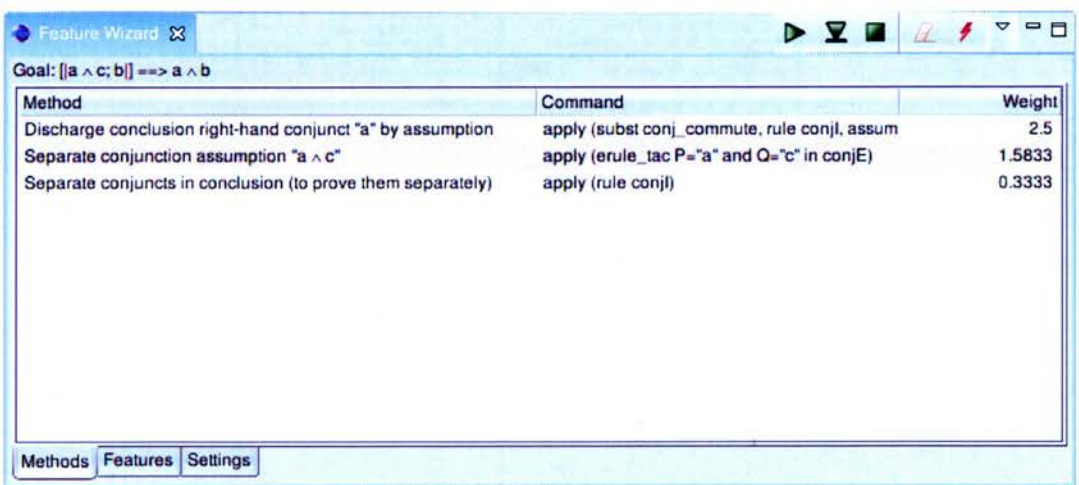

FIGURE 5.1: Feature Wizard Suggestions for Conjunctions

On a slow machine, a user may observe that the "Separate conjunctions in assumption" appears in the list of methods first; however, when the "Discharge conclusion right-hand conjunct" command is eventually cued, with a weight of 2.0, it appears higher in the list. If the user double-clicks this highest-ranked method, it is inserted in the proof file and sent to the prover, resulting in the revised goal "[a ∧ c ; b] ==> a". Feasch makes only one suggestion for this goal, "Separate conjunction assumption "a ∧ c"" (which had also been made before), and after that command is sent, it detects that the resulting goal can be proved by assumption. The complete proof is now included in our theory file:

```
lemma "[| a & c ; b |] ==> a & b"
apply (subst conj_commute, rule conjI, assumption)
apply (erule_tac P="a" and Q="c" in conjE)
apply assumption
done
```

If the user began instead with the lowest-ranked suggestion, "Separate conjunctions in conclusion", two subgoals would have resulted. When there are multiple subgoals, Feasch acts on the first one by default, like Isabelle, although a developer could write a detector that looks at the goal in its entirety (and users can defer the subgoal, directing Isabelle and Feasch to focus on a different one). Here, initially selecting the lowest-ranked suggestion still leads to a successful, albeit longer, proof; in fact, following any path of Feasch's suggestions will lead to a successful proof in this example. For more complex problems, of course, this will not be the case. The control knowledge will always be incomplete, and the ultimate analysis has to be the user's. The Feature Wizard is a proof assistant; the control knowledge expressed in Feasch will hopefully be useful, for suggesting commands and for minimising typing, but just as simp can do undesirable things, there is no guarantee that its suggestions are good ones.

More Heuristic Control Knowledge

The caveat just noted can actually be very beneficial, because it means that a developer can encode rough-and-ready heuristic control knowledge as well as provably correct decision procedures. Furthermore, weights provide a measure of confidence, and even a very low value results in a suggestion to the user; placed near the bottom of the list, they do not hide "better" suggestions, but they are still available if these higher-ranked commands are not actually appropriate. Likewise, on the execution side, low weights can be given to "long-shot" detectors so that they do not hinder the "better" detectors, but they will be run eventually.

With this in mind, we can write control strategies for finding less guaranteed or more specialised applications of the other rules: implication, universal quantification, and existentials. We saw in §5.1.1 that some uses of these rules are highly desirable. Each is also frequently used in other ways, where they may not be guaranteed (like separating a conjunction in the conclusion), or where the guarantee may be uncommon but very useful when it is found (like separating a conjunction conclusion and proving one conjunct).

When an implication appears as an assumption, for example, a user will sometimes try to prove the implicant (the left-hand side of the implication) in order to obtain the consequent (the right-hand side) as an assumption. If an implication assumption is found, a further check could be performed — as a long-shot — to check whether the implicant matches another assumption; if this long-shot pays out, it is easy, good, and safe to apply the commands to replace the implication assumption by the consequence on its own. The features and schemas which achieve this are:

```
feature "some assumption is an implication" ["P","Q"]
    (some assumption is "?P --> ?Q");

add_top_level_schema (FeatureDetector "some assumption is an implication");

mapping "some assumption is an implication" ["P","Q"]
    [ (NamedProofCommand ("Try proving ?P and thus obtain
          ?Q", "apply (erule_tac P=?P and Q=?Q in impE)"), 0.2),
      (FeatureDetector "one assumption matches
    the implicant of another assumption",
          0.8) ];

feature "one assumption matches the implicant of another assumption" ["P","Q"]
    ((some assumption is "?P --> ?Q") AND (some assumption is "?P"));

mapping "one assumption matches the implicant of another assumption" ["P","Q"]
    [ (NamedProofCommand ("Obtain ?Q by implication from assumption ?P",
          "apply (erule_tac P=?P and Q=?Q in impE, assumption)"), 1.2) ];
```

A similar arrangement of features and schemas can be used for existential conclusions. The existential could be instantiated with a schematic variable which will be replaced by a value at a later time; this may be inelegant, but it is a supported and sometimes effective way of proving a theorem. Alternatively, the user could choose (in the Feature Wizard's right-click context menu) to insert the command into the proof script file without sending it to Isabelle. She could then replace the schematic variable by the instantiation she wants, without having to type the rest of the command. A long-shot analysis can be done, again as a lower priority task than the top-level schemas, to check whether the body of the existential is matched in any assumption. If an assumption matches exactly, the subgoal can be discharged. More often, the body of the existential might match an expression contained in some assumption; in this case, there is no guarantee that it is the right instantiation, but it frequently is in practice, and it can be useful to have it easily available.

```
feature "the conclusion is an existential" ["P"] (the conclusion is "EX x. ?P
    x");

add_top_level_schema (FeatureDetector "the conclusion is an existential");

mapping "the conclusion is an existential" ["P"]
    [ (NamedProofCommand ("Instantiate existential in conclusion",
          "apply (rule_tac P=?P and x=\"?x\" in exI)"), 0.15),
      (FeatureDetector "the existential conclusion is matched by an assumption",
          0.7),
      (FeatureDetector "the existential
    conclusion is contained in an assumption",
          0.3) ];

feature "the existential conclusion is matched by an assumption" ["Q","X"]
    ((the conclusion is "? x. ?Q x") AND (some assumption is "?Q ?X") );

mapping "the existential conclusion is matched by an assumption" ["Q","X"]
    [ (NamedProofCommand ("Discharge the goal,
    using ?X to instantiate existential
          in conclusion",   "apply (rule_tac P=?Q and x=?X in exI, assumption)"),
          2.3) ];
```

```
feature "the existential conclusion is contained in an assumption" ["Q","X"]
    ((the conclusion is "? x. ?Q x") AND (some assumption contains "?Q ?X")
         AND (instantiated "?X") );

mapping "the existential conclusion is contained in an assumption" ["Q","X"]
    [ (NamedProofCommand ("Use ?X to instantiate existential in conclusion",
         "apply (rule_tac P=?Q and x=?X in exI)"), 0.55) ];
```

The code for the final feature above requires that the quantity ?X be instantiated; this is a useful check to ensure that the assumption being matched is grounded. Although this can fail to make some suggestions which would be right, and it can make some suggestions which are wrong, our intuition was that this would be the right balance.[28]

Universal quantification assumptions are, in many ways, similar to existential conclusions: the challenge is to find the correct instantiation. Our control knowledge can start by suggesting low-ranked schemas for performing the instantiation with a schematic variable, both non-destructively and destructively. It can also queue a task for seeing whether the conclusion matches the body of the quantification, in which case a suitable command can be formed for discharging the subgoal, or a part of the quantification, in which case a suggestion for the instantiation is obtained. Double quantifications are common in practice, so we include control knowledge to analyse whether the conclusion by abstracting two arguments or (because our features require recommendations to be instantiated) even by abstracting only the second argument. As this computation can be expensive — but is occasionally very useful — it is cued with a low weight. A higher number of multiple quantifications could be handled similarly, or by writing a more general detector, but we have not done so simply because those cases are much less common (and, as a practice we recommend whenever the heuristics might be extended, a comment to this effect is included in the theory file where this is defined).

```
feature "some assumption is a forall quantification" ["P"]
    (some assumption is "ALL x. ?P x");

add_top_level_schema (FeatureDetector "some assumption is a forall
    quantification");

mapping "some assumption is a forall quantification" ["P"]
    [ (NamedProofCommand ("Instantiate quantification assumption ?P",
         "apply (frule_tac P=?P and x=\"?x\" in spec)"), 0.18),
      (NamedProofCommand ("Instantiate and remove quantification assumption ?P",
         "apply (drule_tac P=?P and x=\"?x\" in spec)"), 0.16),
      (FeatureDetector "a quantification
    assumption contains the conclusion", 0.5),
      (FeatureDetector "some assumption is a double forall quantification", 0.4)
    ];

feature "a quantification assumption contains the conclusion" ["P","Q","X"]
    ( (some assumption is "ALL x. ?P x") AND (the conclusion is "?Q ?X")
      AND (instantiated "?X") AND (NOT (expression "?Q" is "%x. x"))
      AND (expression "?P ?X" contains "?Q ?X") );
```

[28] This underscores an important point about how the Feasch with Isabelle system can be used. We as developers did not have to — and did not want to — consider all the cases where some features or mappings might be cued. The Feasch model here was not intended as an exhaustive analysis or a decision procedure, but rather a quick-and-dirty expression of our intuition about when certain commands in a certain theory might be used. An attraction of this approach, in our view, is that developers can rapidly capture and share the heuristic knowledge about a domain, in addition to the purely formal definitions that Isabelle otherwise allows. Even if the heuristics are sub-optimal, they are better than nothing, and there is nothing to stop another user from exploring, refining, and customising the heuristics so created. Such "open" community development could lead to a consensus set of the best heuristic control knowledge or even — if available — a complete decision procedure.

```
mapping "a quantification assumption contains the conclusion" ["P","Q","X"]
    [ (NamedProofCommand ("Use ?X to instantiate quantification assumption ?P",
         "apply (frule_tac P=?P and x=?X in spec)"), 0.88),
      (NamedProofCommand ("Use ?X to instantiate (and remove) quantification
         assumption ?P", "apply (drule_tac P=?P and x=?X in spec)"), 0.86) ];

feature "some assumption is a double forall quantification" ["P"]
    (some assumption is "ALL x y. ?P x y");

mapping "some assumption is a double forall quantification" ["P"]
    [ (FeatureDetector "the conclusion is contained in a double forall
         quantification assumption", 0.4),
      (FeatureDetector "the conclusion is buried in a double forall
         quantification assumption", 0.4) ];

feature "the conclusion is contained in a
    double forall quantification assumption"
    ["P","Q","X","Y","Ps","Pt"]
    ( (some assumption is "ALL x y. ?P x y") AND (the conclusion is "?Q ?X ?Y")
      AND (instantiated "?X") AND (instantiated "?Y")
      AND (expression "?P ?X ?Y" contains "?Q ?X
    ?Y") (expression "?Ps" is "%x. ALL y. ?P x y")
      AND (expression "?Pt" is "?P ?X")

mapping "the conclusion is contained in a
    double forall quantification assumption"
    ["P","Q","X","Y","Ps","Pt"]
    [ (NamedProofCommand ("Use ?X then ?Y to instantiate double quantification
         assumption ?P", "apply (frule_tac
    P=?Ps and x=?X in spec, rotate_tac -1,
         drule_tac P=?Pt and x=?Y in spec)"), 0.88),
      (NamedProofCommand ("Use ?X then ?Y to instantiate (and remove) double
         quantification assumption ?P", "apply (drule_tac P=?Ps and x=?X
         in spec, rotate_tac -1,   drule_tac P=?Pt and x=?Y in spec)"), 0.86) ];

feature "the conclusion is buried in a double forall quantification assumption"
    ["P","Q","X","Y","Ps","Pt"]
    ( (some assumption is "ALL x y. ?P x y") AND (the conclusion is "?Q ?Y")
      AND (instantiated "?Y") AND (expression "?P ?X ?Y" contains "?Q ?Y")
      AND (expression "?Ps" is "%x. ALL y. ?P x y") AND (expression "?Pt" is "?P
    ?X") );

mapping "the conclusion is buried in a double forall quantification assumption"
    ["P","Q","X","Y","Ps","Pt"]
    [ (NamedProofCommand ("Use ?X then ?Y to instantiate double quantification
         assumption ?P", "apply (frule_tac
    P=?Ps and x=?X in spec, rotate_tac -1,
         drule_tac P=?Pt and x=?Y in spec)"), 0.88),
      (NamedProofCommand ("Use ?X then ?Y to instantiate (and remove)
         quantification assumption ?P", "apply (drule_tac P=?Ps and x=?X
         in spec, rotate_tac -1, drule_tac P=?Pt and x=?Y in spec)"), 0.86) ];
```

Being Less Formal

To finish on a simpler note, let us turn to applying the substitution rule, "`[| s = t ; P s |] ==> P t`". This is one of the most commonly used rules in Isabelle, and what we have encoded here barely scratches the surface. If we have success with this, there is a wealth of control knowledge — such as many of the `simp` rules — which could be encoded in Feasch. But there is one important aspect of using features we have not touched upon: the psychological theories in Chapter 1 highlighted the vital role played by similarity. So far, we have looked primarily at matching and formal unification, and while this is sufficient for the present purposes, there is little evidence that humans reason like this. It is much more likely that approximate representations of terms, *cf.* feature vectors (§2.2.2.i), play some role in deciding whether to pursue some potential matching.

The use of Java could be helpful here, as there is a wealth of relevant code available (*e.g.*, for fuzzy searching on-line), and as we showed in the previous chapter, feature detectors can equally well be written on the Java side. We will not implement this here due to the complexity of such a systems integration, although we continue the discussion in §9.1.2. As a rough approximation, however, still using ML we can perform one type of similarity by saying that an assumption and a conclusion are "similar" if they can unify with "?P ?s" and "?P t", respectively, where ?P is non-trivial and, optionally, ?s and ?t satisfy some size constraints.

This type of similarity suggests the use of the substitution rule, because if we can establish the above, proving our goal is tantamount to showing "?s = ?t".

```
feature "an assumption almost matches a
    conclusion, needing only an equality proven"
    ["P","s","t"]
    ( (conclusion is "?P ?t") AND (NOT (expression "?P" is "%x. x")) AND
      (some assumption is "?P ?s") AND (instantiated "?s") AND (instantiated
    "?t"));

add_top_level_schema (FeatureDetector "an assumption almost matches a
    conclusion, needing only an equality proven");

mapping "an assumption almost matches a
    conclusion, needing only an equality proven"
    ["P","s","t"]
    [ (NamedProofCommand ("Assume ?s = ?t and use substitution",
          "apply (rule_tac P=?P and s=?s in subst)"), 0.1),
      (FeatureDetector "an equality assumption makes the conclusion match an
          assumption", 0.6),
      (FeatureDetector "an equality assumption makes the assumption match the
          conclusion", 0.6) ];
```

Showing that "?s=?t" may be quite tricky, and applying subst is unpromising on this basis alone. There is the possibility, however, that other detectors (possibly qualitative or on the Java side) could give more support for this substitution. There will be times when the substitution is correct, and if the user has not noticed it, or if the user does not want to type it all in, this suggestion might be useful. It has, however, the lowest weighting we have used so far, 0.1.

As hinted by the previous paragraph and by the final two mappings above, there will be times when the substitution is supported by the problem. We could check whether the desired equality is contained anywhere in the assumptions, as we did for universal quantification, but let us here confine our analysis to the case when the equality exactly matches one of our assumptions (possibly with commuting, rule sym). When this happens, it is almost universally a good idea to discharge the subgoal by a particular sequence of commands:

```
feature "an equality assumption makes the conclusion match an assumption"
    ["P","s","t"]
    ( (conclusion is "?P ?t") AND (NOT (expression "?P" is "%x. x"))
      AND (some assumption is "?t = ?s") AND (some assumption is "?P ?s")
      AND (instantiated "?s") AND (instantiated "?t"));

mapping "an equality assumption makes the conclusion match an assumption"
    ["P","s","t"]
    [ (NamedProofCommand ("Discharge the conclusion by substituting assumption
          ?s = ?t after symmetry",   "apply (rule_tac P=?P and s=?s in subst,
          ?s = ?t after symmetry", rule sym, assumption, assumption)"), 2.0) ];

feature "an equality assumption makes the assumption match the conclusion"
    ["P","s","t"]
    ( (conclusion is "?P ?t") AND (NOT (expression "?P" is "%x. x"))
```

```
      AND (some assumption is "?s = ?t") AND (some assumption is "?P ?s")
      AND (instantiated "?s") AND (instantiated "?t"));
mapping "an equality assumption makes the assumption match the conclusion"
    ["P","s","t"]
    [ (NamedProofCommand ("Discharge the conclusion by substituting assumption
         ?s = ?t",   "apply (rule_tac P=?P and s=?s in subst,
         assumption, assumption)"), 2.0) ];
```

Finally, though it is not *sim*ilarity, the rule `sym` is frequently useful on its own:

```
feature "an assumption matches the equality conclusion by symmetry" []
    ((conclusion is "?s = ?t") AND (some assumption is "?t = ?s"));
add_top_level_schema (FeatureDetector "an assumption matches the equality
    conclusion by symmetry");
mapping "an assumption matches the equality conclusion by symmetry" []
    [ (NamedProofCommand ("Discharge the conclusion by symmetry to an
         equality assumption",   "apply (rule sym, assumption)"), 2.0) ];
```

5.2. Causal Information for Natural Language Proofs (EH1)

Although we hope that the control knowledge described in §5.1 is useful, our primary intent has been to use it as a means for exploring how control knowledge for Isabelle can be expressed in Feasch. One observation that has struck us is that it inherently makes available the reasons why suggestions were made. In the conjunction example, Feasch found the feature "`the conclusion is a conjunction`", and because of this, it looked for the feature "`the right-hand conjunct of the conclusion matches an assumption`" and found it as well, and this was the reason it suggested to "`discharge right-hand conjunct of conclusion by assumption`". This provides some insight into why suggestions were made, it seems, and where a suggestion is used — or where it matches the proof step the user makes, even if the user did not use Feasch was not used to — this information could, we hypothesise, be used to provide an explanation for proof steps.

There has been a great deal of interest in developing automated systems for natural language generation (NLG) of proofs. The most advanced of these — P.Rex (Fiedler, 2001), Theorema (Buchberger *et al.*, 2006), and ClamNL (Alexoudi *et al.*, 2004) — allow the user to create descriptions at various levels of detail, and higher-level text can be quite successful at removing the uninteresting low-level steps raised as a criticism in §4.2.2.i, often creating good readable proofs in some domains:

> "**Theorem.** $(x + (y + z)) = ((x + y) + z)$ for all natural numbers x, y, and z
>
> "The proof is by one-step induction on x. In order to complete the step case, the conjunction is generalised by introducing a universal variable, *i.e.*, k. The generalised conjecture can be proven by one-step induction on y and finally the proof is completed."
>
> (*ibid.*)

However, there are two limitations noted for these systems and others. Firstly, the text is normally available only for a few domains where natural language support was explicitly developed, presumably because there is a significant amount of work necessary to be able to create such natural language descriptions.

Secondly, despite the stated intention of many of these systems to produce "explanations", the attempts "mostly focus in resembling the way that mathematicians write their proofs, rather than the way that mathematicians reason during proof construction" (*ibid.*). We suspect that this is because the human reasons are absent from the resulting proofs, and plausible reasons are difficult to reconstruct for the automation techniques typically used in proof assistants. The tactics `simp` and `auto`, for example, rely on complex decision procedures which, although very powerful, do not lend themselves to inspectability. The only output these tactics normally produce is the transformed goal; any indication of what they did — never mind why — is buried in the usually enormous trace of a brute-force search. IsaPlanner, we suspect, could be annotated to provide descriptions on multiple levels, because it is based on many of the same principles as the proof-planning system on which ClamNL acts, but as with ClamNL, this information will not include motivation for pursuing steps unless this is added *ad hoc* for every plan step.

In contrast to these typical techniques, the causal information seems to be available for Feasch. To try to establish whether Feasch is, in fact, different to other techniques, we have defined our first "expressivity hypothesis":

> **(EH1)** Features provide reasoned explanations for method selection, *e.g.* for use in generating natural language proofs with causal information.

If we are able to transform the information made available by Feasch into a natural language proof which includes these explanations, we will succeed in showing one way that this technique differs from others.

As a relatively simple attempt to address this question, we have devised a system for stringing together the features that cued a schema, following the activation chain in the mapping network. Since we have used complete sentences for each feature, and imperative sentences for command schemas, we can use a template of natural language connectives to combine the sentences to produce the explanation, with the following constituents (the syntax "`a [ b | c ]`" means to follow `a` by `b` or `c`):

`START1(`f`)` := ["because f" | "observing that f" | "as f"]

`START2(`f`)` := ["we noticed that f" | "we could see that f" | "we found that f"]

`SUBSEQ1(`f`)` := " and " [`START1(`f`)` | "f"]

`SUBSEQ2(`f`)` := " and " [`START2(`f`)` | "that f"]

`NESTED11(`f`)` := " and then " ["that f" | `START1(`f`)`]

`NESTED12(`f`)` := [", we checked and found that f" | ", we then observed that f"]

`NESTED21(`f`)` := [" and then " `START1(`f`)` | "; next, " `START1(`f`)`]

`NESTED22(`f`)` := [" and then that f" | "; as a result, " `START2(`f`)`]

`COMMAND1(`σ`)` := [", our next step was to σ", ", it seemed like a good idea to σ"]

`COMMAND2(`σ`)` := [", so we chose to σ", " and that is why we wanted to σ"]

Using these elements, our process is as follows:

Let σ be the schema which we are explaining. Let $M^{(0)}$ be the ordered set of features which gave positive cues directly to σ, ordered by these cues decreasing. For $i \in Z$, let $M^{(i+1)}$ be the set of features f which cue the detectors that found features in $M^{(i)}$ such that $\forall j \leq i . f \notin M^{(j)}$; let $|M^{(i)}|$ denote the number of elements in $M^{(i)}$ and define h as the largest i such that $M^{(i)}$ is non-empty. Finally, let $\{M_0^{(i)}, M_1^{(i)}, \ldots, M_{|M^{(i)}|-1}^{(i)}\}$ denote the elements in $M^{(i)}$ in order. Define two successor-word functions as follows:

$$\texttt{SEQ1}(M_n^{(i)}) \texttt{ :=} \begin{cases} \texttt{SUBSEQ1}(M_{n+1}^{(i)})\ \texttt{SEQ1}(M_{n+1}^{(i)}), & (n+1) < |M^{(i)}|; \\ \texttt{NESTED11}(M_0^{(i-1)})\ \texttt{SEQ1}(M_0^{(i-1)})\ \texttt{|} & \\ \texttt{NESTED12}(M_0^{(i-1)})\ \texttt{SEQ2}(M_0^{(i-1)}) \texttt{ ,} & (n+1) \geq |M^{(i)}| \wedge i > 0; \\ \texttt{COMMAND1}(\sigma), & (n+1) \geq |M^{(i)}| \wedge i = 0. \end{cases}$$

$$\texttt{SEQ2}(M_n^{(i)}) \texttt{ :=} \begin{cases} \texttt{SUBSEQ2}(M_{n+1}^{(i)})\ \texttt{SEQ2}(M_{n+1}^{(i)}), & (n+1) < |M^{(i)}|; \\ \texttt{NESTED21}(M_0^{(i-1)})\ \texttt{SEQ1}(M_0^{(i-1)})\ \texttt{|} & \\ \texttt{NESTED22}(M_0^{(i-1)})\ \texttt{SEQ2}(M_0^{(i-1)}) \texttt{ ,} & (n+1) \geq |M^{(i)}| \wedge i > 0; \\ \texttt{COMMAND2}(\sigma), & (n+1) \geq |M^{(i)}| \wedge i = 0. \end{cases}$$

The explanation for σ was chosen is then given by:

$$\texttt{START1}(M_0^{(h)})\ \texttt{SEQ1}(M_0^{(h)})\ \texttt{|}\ \texttt{START2}(M_0^{(h)})\ \texttt{SEQ2}(M_0^{(h)})$$

Applying it to the conjunction lemma and the proof given above (and setting upper and lower case appropriately), gives the following sample explanations (shown in *italics*):

```
lemma "[| a & c ; b |] ==> a & b"
  [| a ∧ c ; b |] ==> a ∧ b
```

Observing that the conclusion is a conjunction and then because the right-hand conjunct of the conclusion matches an assumption, it seemed like a good idea to discharge right-hand conjunct of conclusion by assumption.

```
apply (subst conj_commute, rule conjI, assumption)
  [| a ∧ c ; b |] ==> a
```

We could see that some assumption is a conjunction so we chose to separate conjunction assumption $a \wedge c$.

```
apply (erule_tac P="a" and Q="c" in conjE)
  [| b; a ; c |] ==> a
```

Because an assumption matches the conclusion, our next step was to apply assumption.

```
apply assumption
  No subgoals.
done
```

The explanation is readable and provides causal information about the selection of the various steps. The study confirms the hypothesis; furthermore, no domain-specific knowledge or coding was necessary to create this output, so the approach can be expected to apply to any area of mathematics where features can be used. We require that features and command schemas be encoded with our convention, that features are sentences about the problem state and command schemas are named in the imperative mood, but we would recommend doing this anyway for general readability reasons. With the very primitive semantics we use, and with the omission of articles in certain schema names, the output text is far from winning literary awards, but there are

a host of NLG techniques which could be brought to bear on these problems; this is intended as a proof-of-concept, and as such it proves our hypothesis.

It is important to highlight one philosophical distinction about causation: the reasons given by this system are the efficient cause for a step, *i.e.*, what led to its occurrence; this contrasts with the "explanations" typically given by other systems, which are — in all cases that we know of — restricted to the final cause, *i.e.*, why a step works. As the final cause is determined wholly by the proof, there is nothing in our approach to obstruct our doing that analysis here, although given the sophistication of some NLG proof interfaces — both in terms of semantics and navigability, *e.g.*, by expanding and collapsing nodes to read proofs at varying levels of detail — it would be far more promising to extend a pre-existing system, using the additional information provided by Feasch to enrich the proof discourse. This could have particularly welcome applications in interactive or tutorial systems; for although mathematicians are often happy to act at the level of final cause, students require the efficient cause, crucially, to develop their intuition and understanding. A button labelled "How'd ya figure out to do that?" is lacking from nearly all systems; our system provides these reasons inherently.

What is noteworthy, from the perspective of our hypothesis, is that it is straightforward using Feasch to get not only causal information about selection, but also human-readable natural language explanations of that information, *i.e.*, why a step was chosen. Neither of these prospects are readily available from most other control knowledge techniques, and certainly not currently available in Isabelle; we have validated (EH1), and in so doing, we have identified one marked difference in the expressivity of control knowledge in our approach *vis-à-vis* others.

5.3. Interactive Theorem Proving (EH2)

The example in §5.1 demonstrated how Feasch can be used to guide interactive theorem proving. We noted in §4.2.1.i that the tactics in Isabelle are powerful automation techniques but are not interactive. This, in our view, is not merely because people have not tried but because the coding of tactics usually involves a brute force search or a decision procedure. The Isabelle mailing list periodically receives requests from users wanting to see what is going on under the hood with tactics (often, to see what is going wrong), but the behaviour of these tactics is usually too cumbersome even to be inspectable, and an attempt to use them for interactive problem solving will be replete with difficulties.

IsaPlanner, we have noted, addresses some of these limitations by using proof planning ideas with the type of programming constructs usually used in tactics. User interaction with IsaPlanner is possible, using a command-line interface, and where an appropriate plan is available and found by the user, this interface can be a successful means of guiding search in the framework of the plan. There is significant user overhead in choosing to start an IsaPlanner session, however, and the user must restart this session if he wants to switch between plans. On a practical level,

there are not yet many plans available, aside from rippling for inductive proof which works very well on that class of problems, On a theoretical level, even if there was a large library of plans, the fact that a specific plan must be invoked by the user for a specific problem leads to an inflexible user experience; in practice, control knowledge for different domains will need to be combined or interleaved, but the only way to do this in IsaPlanner would be to write a new plan specifically for the combination of these domains.

In the previous chapters and this one, we have observed that features seem potentially quite useful for capturing pieces of control knowledge which can be combined and re-used as necessary. Let us now test this hypothesis:

(EH2) Features enable useful guidance for interactive theorem proving.

5.3.1. The Graphical User Interface (GUI): "Point-and-Click"

Before we describe our experiment, we should summarise the components contributed by "Feasch with Isabelle" to the Eclipse ProofGeneral IDE (integrated development environment). The standard ProofGeneral user interface provides two main widgets: a text editor where commands can be typed and sent to the theorem prover, and a window where the prover's output appears. Feasch with Isabelle provides one more view, with tabs to show either the features found in the current problem or the current ranked list of suggested command schemas.

In interactive mode, the detectors run automatically on any new proof state and the results are updated in the view in real-time. Double-clicking any method will suspend Feasch's evaluation and send the chosen command both to the editor and to the prover. Right-clicking the items gives a context menu with more possible actions, *e.g.*, inserting a command into the editor without sending it to the prover. There are, of course, many ways this interface could be improved; however, we note that this seems to be the case for every IDE in history! Many of these ideas are noted in §9.1.1 for future work, but one vital feature which is present is that it can be turned off or ignored, and will then have no impact on the user's theorem proving activity.

5.3.2. Utility Evaluation

Even as the system currently stands, it presents ample opportunity for an evaluation of its usefulness. This experiment, in fact, should be undertaken before significant additional work is done, because we must still establish whether the use of features is very different to other techniques. By investigating the applicability of Feasch to interactive theorem proving, we will be looking to find both whether Feasch is successful where other techniques are not (and so it is different in this respect) and whether it is promising enough to merit any further development.

Our evaluation will consist of considering how useful the suggestions are; specifically, for how many proof steps does it eliminate or cut down user typing? Theorems in the real-world are

rarely as simple as "[| a ∧ c ; b |] ==> a ∧ b", so as a lifesize experiment let us look at to what extent our Feasch model is applicable to three more complicated theorems from actual theory files:

```
lemma summable_comparison_test: "[| ∃N. ∀n ≥ N. abs(f n) ≤ g n ; summable g |]
    ⟹ summable f"
lemma Interval_expand: "[| Interval R ; R a ; R c ; a≤b ; b≤c |] ⟹ R b"
lemma FTC_E0_ii: "[| ∀x. R x ⟶ DerivE F x = f x ; DifferentiableOn F R ;
    Interval R ; R t ; R x |] ⟹ IntegralE0 (R,t) f x = F x - F t"
```

The first of these rules is part of the Hyperreal library included with the standard Isabelle distribution (used to define the real numbers). The second and third come from our foundation of a theory of calculus, discussed later in this thesis. We chose these lemmas because they are interesting in their own right, they can cannot be solved by the automation techniques or by simple applications of a few other lemmas (unlike many theorems whose proofs, when formalised, use lemmas which are spectacularly uninteresting but which lend themselves to some of the other tactics). Being able to have interactive guidance on some of the low-level rules is particularly useful for the second lemma, because neither `simp` nor `auto` can be used: they enter endless loops and eventually we have to terminate Isabelle, restart it, re-process our entire theory to return to the same point, and then try to work manually — tediously — from first principles. Each one, of course, was proven in the context of the appropriate theories. We will not give the underlying definitions here because they are all roughly as expected; interested users can find them with Isabelle or in our source code on-line.

Our evaluation will consist of ranking each atomic step in the proof into one of four categories, according to how well the Feature Wizard performs on that step:

Perfect: the step can be taken directly from the top-ranking suggestion, with all variables fully instantiated

Good: one of the top ten ranked suggestions can be used, with any schematic variables left in the suggested proof command being correctly instantiated by Isabelle without any typing by the user

Fair: one of the suggestions can be used to make the step, although schematic variables might need to be adjusted by the user

Fail: either no suggestions were made or none of the suggestions are appropriate

Because we were evaluating the Feasch Feature Wizard as an assistant for interactive proof authoring, we were not concerned with necessarily following the original proofs for each of these theorems. We accepted any suggestions which contributed directly to the goal, but of course we did not accept suggestions which either did not apply or were unnecessary in the proof. As it turned out, the resulting proofs were very similar to the originals, apart from minor alterations in order and the absence of `simp` and `auto` in these proofs.

	Summable	Interval	FTC	*Total*
Perfect	20	17	37	*74*
Good	3	1	3	*7*
Fair	1	1	1	*3*
Fail	14	5	17	*36*
Total	*38*	*24*	*58*	120

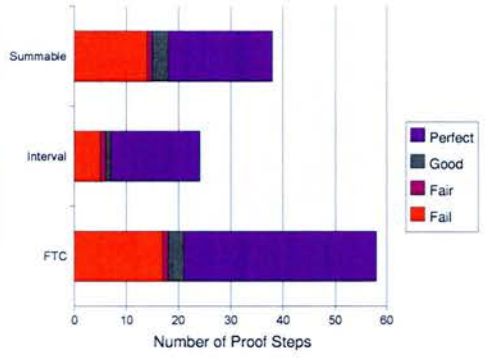

FIGURE 5.2: Feature Wizard Performance on Three Proofs

Figure 5.2 shows the results of this analysis on these proofs of length 38, 24, and 58. The control knowledge we have developed so far gave a perfect answer in the top-ranked suggestion on 61.7% of the proof steps. On a further 8.3% of the steps, the performance was good or fair; the top-ranked suggestion was not appropriate, but another suggestion could be used. The right step or steps was not found for the other 30% of the steps. Feasch indubitably is based on features, and it is helpful at making useful suggestions for interactive theorem proving, so our hypothesis (EH2) has been confirmed.

5.3.3. Discussion

To conclusively show that Feasch is useful, these results must be replicated on a large body of proofs; in fact, even that would not suffice. The only real evidence is a large group of real-world users choosing to use it on real-world problems. From this analysis, however, we have some hope that such a goal might be achieved.

There were several interesting observations about the steps that were and were not suggested by Feasch. Nearly all the "fails" were on commands and rules that we had not programmed in to Feasch. A few of these were integral to HOL, *e.g.*, `disjE`, but most were from more specialised domains, *e.g.*, real arithmetic or calculus definitions. Because the Feature Wizard had no way of knowing about these rules, its failure here, using the simple knowledge base described in this chapter, is unavoidable. If suitable control knowledge were to be encoded for these other domains, we would not be surprised in the 30% fail ratio were decimated. The experience writing heuristics for HOL in Feasch suggests that this may not be very difficult, at least for some instances.

One unexpected result was the low distribution of steps into either the "good" or "fair" category. Although we anticipated the number of "perfects" to be significant due to obvious rule applications such as "`apply assumption`" or "`erule exE`", we were surprised that a relatively large percentage of the steps — around 10% — were found by the "long-shot" detectors we coded.

Furthermore, a cursory look at many of the "goods" and "fairs" suggests easy ways the control knowledge could be extended to lead to "perfect" behaviour. One simple, useful detector would be one which looks for an implication within a universal quantifier assumption, and if it finds it, cues a detector to try to match the conclusion against the consequent of this implication, in order to infer candidate instantiations. This is similar to the existing feature that "`the conclusion is contained in a double forall quantification assumption`", with the useful insight that matches against the left-hand side of an implication will only prove frustrating.

Secondly, we observed good evidence that a weaker notion of similarity would be useful. On several occasions, we saw immediately where the rough matching should be, but Feasch would not suggest it. Looking more closely, we noticed absolute value bars which prevented the matching algorithm from applying: however, in the context of trying to identify a small expression to instantiate a "forall" assumption, when the conclusion has significant structural similarity, and it suggests a candidate expression, our features should overlook such "punctuation" as absolute value bars. This is what we as users did, and although it is not easily done with the current ML code, it may be easier with some of the Java facilities afforded in Feasch.

Logically guaranteed reasons for tactics have been very well studied, and the Feasch approach will never compete with an optimised decision procedure. The control knowledge described here is simple and basic; despite its demonstrated utility, it has been for us little more than an exploration of the capabilities and style of the system. There is a great deal of work which could be done to make the control knowledge for HOL much better. For example, the tactics `auto` and `simp` were used heavily in the original proof of `summable_comparison_test`, and they "fail" on just 11 of the 38 steps. Feasch's performance, failing on 14, is not so shabby, given that we encoded only a small amount of control knowledge, based only on our experience without any complex logical or statistical analysis or attempts at optimisation. What is especially interesting, however, is that Feasch's failures did not coincide with the automated tactics' failures. Approximately half of the places where `auto` and `simp` failed, Feasch proposed a suggestion, often "perfectly". Some of our long-shot detectors, crude as they were and rigid in their view of similarity, were nevertheless capable of succeeding where the well-established, pre-existing techniques failed. By combining the knowledge in both systems — or possibly by implementing the decision procedures in Feasch — a much more powerful system for both automated and interactive proof could be born.

Chapter 6. Automation in Differential Calculus

Having seen that features encode control knowledge in a way useful for interactive theorem proving, we are interested to see how they work as a means for describing fully-automated decision procedures. We should also broaden our focus from low-level logic to higher-level mathematical domains, and in doing so we can also examine the modularity of this means of expressing control knowledge.

After considering many potential areas, we focussed on college calculus as a subject which has real-world familiarity as well as problems which range in difficulty. We were attracted to solving derivatives as a type of problem for which a decision procedure can be applied, and further questions — such as solving integrals — are available if something more challenging is desired. Isabelle has a library of the reals (founded on hyperreals; implemented by Fleuriot & Paulson, 2000) which includes a formulation of the derivative and integral operator.

6.1. A Formal Theory of Calculus

While the theory of the reals is heavily used, we discovered that the calculus operators have not been tested so thoroughly. Many theorems we expected to find were absent, and several months were spent finalising the theory.

The most important theorem which was missing was the first half of the fundamental theorem of calculus. The Hyperreal library includes `FTC1`, roughly that $\int \mathrm{d}f = f \ (+C)$ (this is commonly called the second part of the fundamental theorem, despite the name):

```
lemma FTC1: "[|a ≤ b; ∀ x. a ≤ x & x ≤ b --> DERIV f x :> f'(x) |] ==>
    Integral(a,b) f' (f(b) - f(a))"
```

The first part, that $\mathrm{d} \int f = f$, is not a corollary of this theorem. Proving this requires showing, among other lemmas, that continuous functions are bounded over finite regions. To simplify our proofs, we also created a theory for intervals: "`Interval` R" means that R is an interval, mapping a real argument to a boolean to determine containment; *i.e.*, for any two points a and b such that $R\,a$ and $R\,b$, if $a < x < b$, then $R\,x$ also holds. "`IntervalInside` $R\,x$" means that $R\,x$ holds and that x is on the interior (*i.e.*, there exists a neighbourhood around x where all points are in R). Armed with these concepts and proofs in Isabelle, we developed:

```
lemma FTC_i: "[| ∀ a b. R a & R b & a<b --> Integral(a,b) f (F b - F a) ;
    IntervalInside R x ; ContinuousAt f x |] ==> DERIV F x :> f x"

lemma FTC_ii: "[| ∀ x. R x --> DERIV F x :> f x ; Interval R ; R a ; R b ; a<=b
    |] ==> Integral(a,b) f (F b - F a)"
```

The proof of `FTC_i` is lengthy, and `FTC_ii` is trivial (we use `FTC1`, including it purely for the sake of consistency of notation and naming), so both are omitted here.

One joy encountered in building the proof of FTC_i was the gauge integral. This idea — developed only in the last fifty years — is a generalisation of the integrals of both Riemann and Lebesgue, with a simple and elegant definition. It was used in the original Hyperreal theory definition, and so while it made our work substantially more difficult (*e.g.*, than proofs would be for the common Riemann integral), it means the resulting theories are applicable to a much wider class of functions. Of particular note is the characteristic function χ of the rationals:

$$\chi(x) = \begin{cases} 1, & x \in \mathbb{Q} \\ 0, & \text{otherwise} \end{cases}$$

This is integrable in the gauge sense: the reals dominate, and on any interval $\int \chi(x)\mathrm{d}x = 0$. But $\mathrm{d}[0]/\mathrm{d}x = 0$, and FTC_i, that the derivative of integral any function is that function, has to be qualified with a continuity assumption. Discontinuities also affect other integrals, but if the function is integrable in the Riemann sense, *e.g.*, the equality in the conclusion can be qualified merely by saying "except at the finitely many points where f may be discontinuous". With the gauge formulation, we concluded it would make more sense to add a continuity assumption.

Another extension to the provided library we wished to make is to be able to use the integral and the derivative as operators. The Hyperreal library defines each as a multi-argument predicate which takes both F and f and returns true if and only if $\mathrm{d}F = f$ (with F and f used in the appropriate places). We felt that having the integral and the derivative defined as true morphisms, mapping from real-valued functions to real-valued functions, would be much more useful, but we also wanted to be able to reason about indefinite integrals. This introduced the complexity of either specifying a choice of constant or representing the family of functions; we preferred the former possibility, but as not all functions are integrable over all regions, we opted to introduce a parameter which indicates the region of integrability (R) as a parameter of our integration predicate and the point where the integral should take on a zero value (t). Our definitions are as follows:

```
DerivE :: "(real=>real)=>(real=>real)" "DerivE F == (%x. (THE k. DERIV F x :>
    k))"
IntegralE0 :: "[(real=>bool) * real]=>(real=>real)=>(real=>real)" "IntegralE0 ==
    %(R,t) f. (%x. if (R x) then if (t<=x) then (THE F. Integral(t,x) f F) else
    -(THE F. Integral(x,t) f F) else 0)"
```

Additionally, the THE token is a primitive in Isabelle/HOL which returns the unique value satisfying its argument. Its behaviour can be unintuitive, if, say, some function is not differentiable or integrable. Our definitions and proofs are unfortunately complicated as a result of these issues, but their application and use is rigorously formalised (as required by Isabelle), and happily they are easy to use.

6.2. Automated Theorem Proving (EH3)

The third expressivity hypothesis we will test is whether features can represent the type of control knowledge used in decision procedures and other sophisticated heuristics, and whether we can use this to solve problems automatically.

(EH3) Features can encode control knowledge for automated theorem proving.

Using Feasch in the domain of calculus, we looked specifically at taking derivatives. This is an area where the appropriate control knowledge can automatically find solutions. How easily can we express this control knowledge in Feasch?

6.2.1. A Decision Procedure for Differentiation

The decision procedure for differentiation proceeds by analysing the problem at every step and iteratively applying the rule appropriate for the top-level form of the function. In the current experiment, we exclude implicit and partial differentiation from consideration, staying at the level of a first-year college calculus course; these require more machinery than we wish to develop, including algebraic decision procedures outwith the area of differentiation. The same is true for functions such as $\sqrt{x}$ which are not defined everywhere over $\mathbb{R}$; we include the rules for reducing many such functions, but we do not develop the machinery for reasoning automatically about intervals. Trying to find the derivative of explicitly defined entire functions in a single variable is sufficient for the purposes of formally testing (EH3). We do, however, look at case studies of its performance on other problems.

Our theory of calculus includes proofs for rules for differentiating over each top-level connective (continuity assumptions not shown):

Rule Name	Formula
`deriv_const`	$\mathrm{d}[C]/\mathrm{d}x = 0$
`deriv_id`	$\mathrm{d}[x]/\mathrm{d}x = 1$
`deriv_add`	$\mathrm{d}[f(x) + g(x)]/\mathrm{d}x = f'(x) + g'(x)$, where $f'(x) = \mathrm{d}[f(x)]/\mathrm{d}x$
`deriv_minus`	$\mathrm{d}[-f(x)]/\mathrm{d}x = -f'(x)$
`deriv_diff`	$\mathrm{d}[f(x) - g(x)]/\mathrm{d}x = f'(x) - g'(x)$
`deriv_mult`	$\mathrm{d}[f(x) * g(x)]/\mathrm{d}x = f(x) \cdot g'(x) + g(x) \cdot f'(x)$
`deriv_inverse`	$\mathrm{d}[1/x]/\mathrm{d}x = -1/x^2$
`deriv_div`	$\mathrm{d}[f(x)/g(x)]/\mathrm{d}x = (f'(x) \cdot g(x) - g'(x) \cdot f(x))/g^2(x)$
`deriv_fx_pow_n`	$\mathrm{d}[f(x)^n]/\mathrm{d}x = n \cdot f(x)^{n-1}$, where $n \in \mathbb{N}$ and $n > 0$
`deriv_x_rpow_z`	$\mathrm{d}[f(x)^{g(x)}]/\mathrm{d}x = \mathrm{d}[e^{g(x) \cdot \ln f(x)}]/\mathrm{d}x$, where $f(x) \in \mathbb{R}$ and $f(x) > 0$
`deriv_chain`	$\mathrm{d}[f(g(x))]/\mathrm{d}x = f'(g(x)) \cdot g'(x)$

FIGURE 6.1: Rules for Solving Derivatives

To encode the decision procedure, we need to specify the features when each of these rules is appropriate. For many of them, it is simply a case of recognising the topmost connective in the function being differentiated. It would be very inefficient to check for each top-level connective at every step in every proof; we therefore begin with one feature checking whether a derivative is present:

```
feature "has DerivE" ["f"] (goal has "DerivE ?f");
add_top_level_schema (FeatureDetector "has DerivE");
```

This top-level feature detector will cue all the other features.

Let us now look at each of the two-argument functions. Detecting their presence is straightforward:

```
feature "DerivE of sum" ["f","g"] (goal has "DerivE (%x. (?f x)+(?g x))");
mapping "DerivE of sum" ["f","g"] [ (NamedProofCommand ("deriv_add ?f + ?g",
    "apply (subst deriv_add[of ?f _ ?g])"), 1.5) ];
```

```
feature "DerivE of mult" ["f","g"] (goal has "DerivE (%x. (?f x)*(?g x))");
mapping "DerivE of mult" ["f","g"] [ (NamedProofCommand ("deriv_mult ?f * ?g",
    "apply (subst deriv_mult[of ?f _ ?g])"), 1.45) ];
feature "DerivE of diff" ["f","g"] (goal has "DerivE (%x. (?f x)-(?g x))");
mapping "DerivE of diff" ["f","g"] [ (ProofCommand "apply (subst deriv_diff[of
    ?f _ ?g])", 1.48) ];
feature "DerivE of div" ["f","g"] (goal has "DerivE (%x. (?f x)/(?g x))");
mapping "DerivE of div" ["f","g"] [ (ProofCommand "apply (subst deriv_div[of ?f
    _ ?g])", 1.43) ];
mapping "has DerivE" [] [ (FeatureDetector "DerivE of sum", 1.5),
    (FeatureDetector "DerivE of mult", 1.45), (FeatureDetector "DerivE of diff",
    1.48), (FeatureDetector "DerivE of div", 1.43) ];
```

Often there are many features and detectors possible. We could have instead written a single detector which looks for expressions of the form "`DerivE (%x. (?h (?f x) (?g x)))`" where "`size "?h" (fn s => s=1)`", *i.e.*, `?f` is a single connective, with further tests to ensure `?f` and `?g` are instantiated. The choice of how to write features is left up to the developer; what Feasch requires is that these features — and associated detectors — be made explicit, given names, and left independent of any expected consequential action or rule. Mappings, as shown above, are specified separately to cue the appropriate rules on the basis of features; this is a central difference between Feasch and other control knowledge techniques, and which we believe was crucial to the results shown in (EH1) and (EH2). Here, making the features explicit seems an unusual step — it is not a part of standard control knowledge techniques — but it does not significantly obstruct our development of the differentiation decision procedure. It may be a bit more work, but let us withhold judgment on this issue.

One theoretical advantage of defining the features as above, and not using a single detector which examines the top-level detector (we do have alternate code for this approach), is that it is slightly more general. If the "topmost connective" were not well-defined, as arguably it is not when a person parses an expression (*e.g.*, $A+B+C$), or in a hypothetical system where unification is transparent across associativity, these features would apply to any of the topmost connectors. This is also one of the reasons that slightly different weights for the different rules is used. The other reason is for the case where a goal has several derivative expressions: we simply felt it would be nicer to pick off the "easier" case of differentiating across a sum first. In practice, these weights and choices do not affect our statistical results, and merely illustrate the type of design choices one makes with Feasch.

There are two other notable two-argument connectives, both related to exponentiation. Many theories in Isabelle assume the use of the natural numbers, where exponentiation is defined using the token "`^`". The theory of the Hyperreals defines "real power" using the token "`^^`". In our system, we wish to be able to find derivatives involving either expression:

```
feature "DerivE of pow, int" ["f","n"] (goal has "DerivE (%x::real. (?f x) ^
    ?n)");
mapping "DerivE of pow, int" ["n"] [ (NamedProofCommand ("deriv_fx_pow_n",
    "apply (subst deriv_fx_pow_n[of ?f ?n])"), 1.4) ];
```

```
feature "DerivE of pow, real" ["f", "g"] ((goal has "DerivE (%x. (?f x) ^^ (?g
    x))") AND (NOT (expression "?g" is "%x. ?k")) );
mapping "DerivE of pow, real" ["f", "g"] [ (NamedProofCommand ("deriv_x_rpow_z",
    "apply (subst deriv_x_rpow_z[of ?f _ ?g])"), 1.4) ];
mapping "has DerivE" [] [ (FeatureDetector "DerivE of pow, int", 1.4),
    (FeatureDetector "DerivE of pow, real", 1.3) ];
```

The real power operator is only valid for certain values of $g(x)$, and the underlying rules reflect these requirements, *e.g.*, as shown in Figure 6.1. We have not attempted to account for this in the current automation strategy; proving the underlying requirements is properly a different problem, reasoning about ranges and inequalities, and beyond the scope of the present experiment. We have also defined features and methods for transforming expressions of the form "`DerivE (%x. k)`" into the more easily differentiated form "`DerivE (%x. exp (x * (ln k)))`" for positive `k` (not shown).

With the two-argument connectives all defined, it is time to focus on what is arguably the most powerful part of the decision procedure. Whenever the function being differentiated is the composition of two functions, the chain rule can separate them into their constituent parts. Because our feature detection language uses unification — more flexible than mere matching — there are trivial compositions possible, *viz.*, where one of the functions is instantiated as the identity, so we explicitly disallow these cases.

```
feature "DerivE nested functions" ["f","g"] ((goal has "DerivE (%x.
    ((?f::(real=>real)) (?g x)))") AND (NOT (expression "?f" is "%x. x")) AND
    (NOT (expression "?g" is "%x. x")) AND (NOT (expression "?g" is "%x. ?k")));
mapping "DerivE nested functions" ["f","g"] [ (ProofCommand "apply (subst
    deriv_chain[of ?f ?g])", 1.3) ];
mapping "has DerivE" [] [ (FeatureDetector "DerivE nested functions", 1.3) ];
```

This allows us to use the primitive rules, such as `deriv_is` and `deriv_minus` in Figure 6.1, without needing explicit rules for, *e.g.*, $\mathrm{d}[-f(x)]/\mathrm{d}x$. (Using a two-argument chain rule and partial differentiation, it would have been possible to do the same for the two-argument rules such as addition, but this is rather more difficult.)

With the above control knowledge, we would expect Feasch to be able to reduce any compound expression into elementary parts. We must also, of course, inform the system what rules are applicable for solving particular elementary derivatives. We have proved the relevant results for the functions in our theory; their formulation is as for `deriv_const` and `deriv_id` in Figure 6.1 and is omitted here. The features for these, again, merely detect whether the respective functions are being differentiated:

```
feature "DerivE of const" [] ((goal has "DerivE ?f") AND (expression "?f" is
    "%x. ?k"));
mapping "DerivE of const" [] [ (ProofCommand "apply (subst deriv_const)", 2.0) ];
feature "DerivE of id" [] (goal has "DerivE (%x. x)"); mapping "DerivE of id" []
    [ (ProofCommand "apply (subst deriv_id)", 2.0) ];
feature "DerivE of negation" ["f"] (goal has "DerivE (%x. -(?f x))");
mapping "DerivE of negation" [] [ (ProofCommand "apply (subst deriv_minus)",
    1.6) ];
feature "DerivE of sin" [] (goal has "DerivE sin");
```

```
mapping "DerivE of sin" [] [ (ProofCommand "apply (subst deriv_sin)", 1.6) ];
feature "DerivE of cos" [] (goal has "DerivE cos");
mapping "DerivE of cos" [] [ (ProofCommand "apply (subst deriv_cos)", 1.6) ];
feature "DerivE of exp" [] (goal has "DerivE exp");
mapping "DerivE of exp" [] [ (ProofCommand "apply (subst deriv_exp)", 1.6) ];
feature "DerivE of ln" [] (goal has "DerivE ln");
mapping "DerivE of ln" [] [ (ProofCommand "apply (subst deriv_ln)", 1.6) ];
mapping "has DerivE" [] [ (FeatureDetector "DerivE of const", 2.0),
    (FeatureDetector "DerivE of id", 2.0), (FeatureDetector "DerivE of
    negation", 1.7), (FeatureDetector "DerivE of sin", 1.6), (FeatureDetector
    "DerivE of cos", 1.6), (FeatureDetector "DerivE of exp", 1.6),
    (FeatureDetector "DerivE of ln", 1.6) ];
```

6.2.2. GUI: "Fast-Forward" Automation

Now that our control knowledge has been encoded in Feasch, we can try to use it for fully automated theorem proving. As part of the Feature Wizard, we have included a technique we call " 'fast-forward' automation": it repeatedly tries applying cued methods, in their rank order, until it reaches a state where none apply. This may be the end of the proof, in which case the user can simply continue developing the next lemma in their theory. It might instead be a step where Feasch with Isabelle does not have relevant control knowledge, and the user has to provide the next step, after which automation could be resumed. Or, depending on whether the control knowledge encoded is safe, it might be the case that the automation technique has pursued a blind alley, the proof state is false, and the user must manually backtrack and send the proof in a different direction.

The "fast-forward" technique does not do any state-caching or backtracking. Search is limited to a local best-first strategy, hard-coded in the Java user interface; if a cued method fails, our technique will try the next on the list, but if all methods in a state fail, the technique stops. To do more requires significant work on the Isabelle ML side and on the GUI side and is not directly relevant to our present enquiry. It would, of course, be a very useful feature, and ideas for pursuing this, connected with IsaPlanner, are reviewed in §9.1.4.ii.

As an example, let us use this automation technique, with the control knowledge as described above, to solve the problem of evaluating $\mathrm{d}[(\sin x)^3]/\mathrm{d}x$. We invoke the "Fast-Forward" button (a downward-pointing triangle with a line underneath, shown in Figure 6.2); it detects the feature "`has DerivE`" and then the feature "`DerivE nested functions`", and a single candidate command, "`apply (subst deriv_chain[of "λu. u ^ 3" "sin"])`". The automation tries this command but then stops, showing a red error icon next to this command.

Our algorithm does not work! By right-clicking the command to insert it in our proof, and sending it manually, we discover that it does not apply. Positioning the mouse over the `deriv_chain` rule in our proof, we can see the Eclipse ProofGeneral "hover help text" which we developed (§4.2.3.ii). This shows us that our version of the chain rule only applies when the derivative is being evaluated at a specific point, because for the chain rule to be valid,

the constituent functions must be shown to be differentiable at that point. We must apply an extensionality rule to transform the current functional equality goal, "DerivE (%. (sin x)3) = ?f", to an equality at a universally quantified point z, "DerivE (%. (sin x)3) z = ?f z".) We create a lemma which helps for this[29] and features to suggest it when appropriate.

```
lemma extAll: "(A=B) = (ALL x. A x = B x)"
apply auto
apply (rule ext)
apply (drule spec[of "%x. A x = B x"])
apply assumption
done

ML {*
feature "DerivE lacks quantification" ["df"] (((goal has "DerivE ?f") AND (NOT
    (goal has "(DerivE ?f) ?x"))) AND (expression "DerivE ?f" is "?df"));
mapping "DerivE lacks quantification" ["df"] [ (ProofCommand "apply (subst
    extAll[of ?df])", 1.5) ];
mapping "has DerivE" [] [ ((FeatureDetector "DerivE lacks quantification"), 1.5)
    ];
*}
```

The mapping specifies that the feature "DerivE lacks quantification" is only checked if a derivative is present in the goal. Looking at the deriv_chain rule, we remarked that it also requires proving certain differentiability assumptions. Although we could use Feasch for this, the control knowledge model for proving differentiability very closely follows the model for solving derivatives. These goals are typically uninteresting and easy to discharge automatically, so we will rely on a decision procedure we developed using a "simpset" for Isabelle's built-in simplifier. We will rely on the Feasch model to use features can detect a differentiability conclusion — or a continuity conclusion because that can often be solved by the same simpset — and cue the appropriate simplification set of rules. Here, the Feasch approach means it is not necessary to add these rules to the standard simplification set; this is a good thing because there are times when we may want to keep differentiability assumptions; and if these rules were used automatically by the simplifier, such assumptions would be removed (and in some cases worse things, such as infinite loops and system crashes, would happen!).

```
feature "differentiability conclusion" ["f"] ((conclusion has "?f differentiable
    ?x") OR (conclusion has "DifferentiableOn ?f ?R"));
add_top_level_schema (FeatureDetector "differentiability conclusion");
mapping "differentiability conclusion" [] [ (NamedProofCommand ("simp (only
    differentiability by smoothness)", "apply (simp only: smooth_manip_ss
    smooth_solve_ss differentiability_by_smoothness_ss)"),
    1.5), (NamedProofCommand ("simp (add differentiability by
    smoothness)", "apply (simp add: smooth_manip_ss smooth_solve_ss
    differentiability_by_smoothness_ss)"), 1.0) ];
feature "continuity conclusion" ["f"] ((conclusion has "continuous ?f") OR
    (conclusion has "ContinuousOn ?f ?R") OR (conclusion has "ContinuousAt ?f
    ?x"));
add_top_level_schema (FeatureDetector "continuity conclusion");
mapping "continuity conclusion" [] [ (NamedProofCommand ("simp (only continuity
    by smoothness)", "apply (simp only: smooth_manip_ss smooth_solve_ss
    continuity_by_smoothness_ss)"), 1.5), (NamedProofCommand ("simp (add
```

[29] There may be a rule already in the library, but we could not find it. Perhaps Feasch control knowledge could have helped with that.

```
continuity by smoothness)", "apply (simp add: smooth_manip_ss
smooth_solve_ss continuity_by_smoothness_ss)"), 1.0) ];
```

With this addition, we try the fast-forward automation again. The system applies `extAll` and then successfully applies the chain rule. After proving the differentiability assumptions, there is a single subgoal, "`∀x. DerivE (λu. u 3) (sin x) * DerivE sin x = ?f x`". The Feature Wizard shows the following for features and methods:

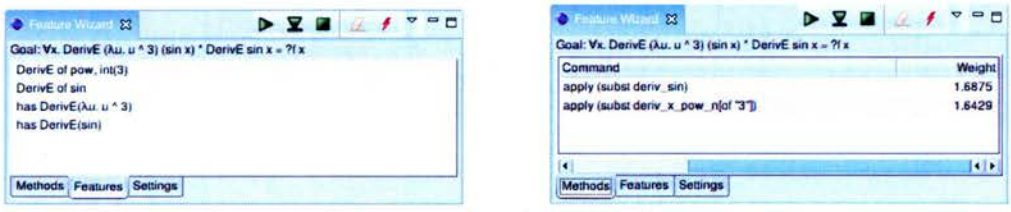

FIGURE 6.2: Fast-Forwarding through Differentiation

The fast-forward automation continues and the following proof is found:

```
lemma "DerivE (%x. (sin x)^3) = ?f"
apply (subst extAll[of "DerivE (?x. sin x ^ 3)"])
apply (subst deriv_chain[of "?u. u ^ 3" "sin"])
apply (simp only: smooth_manip_ss smooth_solve_ss
        differentiability_by_smoothness_ss)
apply (simp only: smooth_manip_ss smooth_solve_ss
        differentiability_by_smoothness_ss)
apply (subst deriv_sin)
apply (subst deriv_fx_pow_n[of "%x. x" _ "3"])
apply (subst sym[OF extAll], rule refl)
done
```

By using the schematic variable `?f` in the equality, after completing the proof, Isabelle instantiates `?f` as "`3 * sin x (3 - 1) * cos x`".

6.2.3. Evaluation and Discussion

Evaluation

We evaluated the algorithm we encoded by testing it on a wide range of differentiation questions, including those on a published examination paper from a first-year college calculus course. We excluded problems where functions are not explicitly defined (including inverse functions) or where the domain of the function is less than the entire reals, although we will discuss such problems shortly. Our system was able to solve all the problems in the test set, including, *e.g.*, $\mathrm{d}[2^{\cos x} + 2^{\sin x}]/\mathrm{d}x$ and $\mathrm{d}^2[16 \cdot t^2]/\mathrm{d}t^2$.

There are many systems which solve derivatives, and the basic decision procedure for it is not difficult. As we would hope, we have shown that it is not difficult to do in Feasch either, and we have confirmed (EH3).

Inspectability

We excluded problems where differentiability only holds for certain ranges: *e.g.*, to solve DerivE (%x. x 1/2)) 1, because the derivative of real exponentiation is only defined for positive values, we have to show $1 > 0$. This can be done by automation tactics such as `simp`; as we have noted, users sometimes try this tactic without thinking when faced with a new problem state, so always sending a cue to this method might actually be useful. In case studies where `simp` is added as a method to apply when all others fail (*i.e.*, cued by "`has DerivE`" with a low weight), Feasch with Isabelle is able to solve many such "incomplete" functions.

An especially interesting behaviour occurs, however, when the "fast-forward" automation fails. As we saw when `deriv_chain` could not be applied, a user can inspect the context of the problem solver even if our automation technique cannot be applied. By looking at this, a developer can debug problems with the algorithm and potentially, as we did, make the necessary improvements.

This so-called "useful failure" also occurs when the control knowledge cannot solve a problem. Given "`DerivE (%x. x 1/2)) 1`", for example, our system will proceed to the point where it cannot prove that $1 > 0$, and then stop. A user can manually discharge this goal, or extend the control knowledge so that the goal can be solved automatically, and then continue with the automation. Crucially, the user can observe what the system is doing and why when it stops. This inspectability is available at any point, while it is running and if a user interrupts it; so, if the automation appears to be going off an a tangent, a user can interfere, manually backtrack, and specify an alternate path.

"Useful failure" is not a property of many other automation techniques. If `simp` does not apply — or does not do what is desired — there is no way for a user to repair the behaviour and continue. The same is true for all other tactics in Isabelle, with the exception of IsaPlanner. In the case of IsaPlanner, however, it is difficult to observe what is occurring in real-time, and it is not possible to interrupt the process. It can be told to stop after a fixed number of plan steps, and at that point a user can direct the planning interactively, but she is restricted to applying commands that were specifically in the original plan. With Feasch with Isabelle, a user can take over if the "fast-forward" fails or merely on a whim, and the technique provides information useful for interactive proving. Combining automated and interactive applicability is not a common property of control knowledge representations, and we suggest that it may be one benefit of this approach.

With many decision procedures, there are times when a problem simply cannot be solved. We restricted our problem set to entire functions; decision procedures used elsewhere typically solve a much wider class of problems. This is not because our algorithm is worse than these other algorithms, but rather, we argue, because ours is better. To begin with, note that some differentiation problems are very difficult to solve rigorously. Consider the derivative of $(\sin x)/x$ at $x = 0$: the function is undefined at 0, so the derivative is similarly undefined. If we explicitly

define it to have a value of 1 at that point, the derivative can be computed to be 0, but only after proving continuity. Typical computer algebra solutions ignore these boundary conditions, and will say that the derivative is 0 when it is actually undefined and fail to find it for the analytic continuation![30] Even if these systems did perform the rigorous analysis that a theorem prover offers (indeed, requires) and could detect the problem, the way automated decision procedures are typically implemented does not facilitate a user's interacting to help on failure. In the case of the analytic continuation, or for differentiating such functions as $\sqrt{x}$, our approach allows a user to manually prove the requirements that the decision procedure cannot handle.

Continuing after Failure

In a fully automated standalone environment, we have implemented a solver which uses this control knowledge and specifically defers any goals it cannot prove. We randomly constructed a set of 100 expressions to test, using the operators and functions in our theory (this corresponds to all functions used in a typical college calculus course). The procedure here finds the derivative to all 100 problems, displaying the assumptions it used which it could not prove, as shown here (k?? stand for arbitrary constants in our randomly created problems):

```
lemma "DerivE (%x. exp (ln (((kUZ)^^(cos (kEy)))-((x)^(kCJ))))) = ?f"
apply (subst extAll[of "DerivE (%x. exp (ln (kUZ ^^ cos kEy - x ^ kCJ)))"])
apply (subst deriv_chain[of "exp" "%u. ln (kUZ ^^ cos kEy - u ^ kCJ)"])
apply (simp only: smooth_manip_ss smooth_solve_ss
        differentiability_by_smoothness_ss)
defer
apply (subst deriv_exp)
apply (subst deriv_chain[of "ln" "%u. kUZ ^^ cos kEy - u ^ kCJ"])
defer
apply (simp only: smooth_manip_ss smooth_solve_ss
        differentiability_by_smoothness_ss)
apply (subst deriv_ln)
defer
apply (subst deriv_diff[of "%u. kUZ ^^ cos kEy" _ "%u. u ^ kCJ"])
apply (simp only: smooth_manip_ss smooth_solve_ss
        differentiability_by_smoothness_ss)
apply (simp only: smooth_manip_ss smooth_solve_ss
        differentiability_by_smoothness_ss)
apply (subst deriv_const)
apply (subst deriv_fx_pow_n[of "%u. u" "kCJ"])
apply (rule allI)
apply (rule refl)
oops

(* ABANDONED PROOF
   ASSUMPTION: !!x. (%u. ln (kUZ ^^ cos kEy - u ^ kCJ)) differentiable x
   ASSUMPTION: !!x. ln differentiable kUZ ^^ cos kEy - x ^ kCJ
   ASSUMPTION: !!x. 0 < kUZ ^^ cos kEy - x ^ kCJ
   PARTIAL ANSWER: lemma DerivE (%x. exp (ln (?kUZ ^^ cos ?kEy - x ^ ?kCJ))) =
          (%x. exp (ln (?kUZ ^^ cos ?kEy - x ^ ?kCJ)) * (inverse
             (?kUZ ^^ cos ?kEy - x ^ ?kCJ) * (0 - real ?kCJ * x ^ (?kCJ - 1))))
 *)
```

It computes that the derivative of $e^{\ln(k_1{}^{\cos k_2} - x^{k_3})}$ is $e^{\ln(k_1{}^{\cos k} - x^{k_3})} \cdot \frac{1}{k_1{}^{\cos k_2} - x^{k_3}} \cdot (-k_3 \cdot x^{k_3 - 1})$, subject to certain assumptions.[31]

Of the resulting solutions, 61 required no additional assumptions; the other 39 all involved division, exponentiation, or logarithms, for which subgoals are understandable. In some instances,

[30] The presence of errors in CA systems is discussed in more detail in §7.1.1.

[31] This is easily verified by using $e^{\ln x} = x$; introducing such "simplification rules" into our Feasch model is an attractive extension.

these subgoals could be proved manually (*e.g.*, $\ln e^x$), but in most of the randomly created problems, the extra assumptions are necessary for a derivative to be found; they indicate the region on which the derivative is valid, and for any solution to be found, equivalent constraints should have been specified in the original problem. This ability to suggest ways of repairing the theorem is a further property not generally true either of techniques in Isabelle or techniques used in computer algebra systems.

Specialisability

Another difference is observed if we attempt to `apply simp` directly to the lemma. The goal is discharged! All that it has done, however, is to instantiate `?f` as the original, unsolved expression, using the rule `refl`. Our use of Isabelle as a calculator is relatively unusual, but it can also be quite powerful. The simplification set is designed with the common cases in mind, but unfortunately it excludes this use. Often, in fact, the simplifier will not be appropriate for a particular problem, but there is no way in Isabelle to specify what type of problems the simplifier should act on.[32] Feasch allows us to specify precisely when control knowledge should be used. In fact, the choice of commands is driven by features noticed, rather than by following a specific tactic or plan script. The use of mappings to establish a hierarchical execution permits specialisation such that only one of the differentiation feature detectors need be run if the problem does not involve `DerivE`. Furthermore, if a user does not want certain detectors or methods to be run automatically, they can easily learn what features and mappings are being used and override them for their situation.

Limitations

While the differentiation decision procedure was successful, it seems that the present Feasch with Isabelle system will have some limitations on more complex decision procedures. The procedure we encoded is "stateless", in the sense that the same algorithm is followed on every proof state. Currently, in Feasch with Isabelle, there is no way to encode a plan step such as $A(B \mid C)$, and no means to preserve features from one proof step to the next. A stateful decision procedure could not be implemented. Secondly, there is no way to backtrack automatically, so if mistakes are made by the "fast forward" automation, the system will not recover; more generally, search is a vital technique for problem solving, both depth-first and breadth-first, and as the simple case of backtracking is not feasible, this poses a severe restriction on its capabilities.

Interestingly, these abilities are fundamental in most other control knowledge techniques, including Isabelle tactics and IsaPlanner plans. The fact that statefulness and search are basic

[32] Actually, it can be done by putting appropriateness conditions as formal assumptions in a simplification rule, but logically this is unattractive, and in practice it has the result that `simp` will — very inefficiently — devote energy to proving the appropriateness conditions.

parts of these techniques, whereas Feasch with Isabelle is quite powerful even without these tools, suggests another way that the current proposed approach differs.

We note, furthermore, that these limitations are not inherent in the Feasch system. In §9.1.4 and §10.1.2, we describe how they can be overcome using the current architecture to implement persistent features and tiered schemas. Finding this proposed solution did, admittedly, require some effort, and these limitations, while not insurmountable, should be recognised as aspects of the Feasch approach which require special attention, possibly in contrast to the other control knowledge techniques.

We write "possibly in contrast" because both tactics and IsaPlanner use specialised code in Isabelle for recording and manipulating proof state graphs. Such a library would also be necessary to have the functionality of allowing statefulness and search in Feasch with Isabelle. Creating this code from scratch would be a major undertaking; a more attractive prospect is the incorporation of these capabilities by integrating it with IsaPlanner, as discussed in §9.1.4.ii.

6.3. Modularity (EH4)

The final expressivity property we will investigate is whether features support modular code development. In good design, the "pieces — the modules — are autonomous, coherent and organised in robust architectures" (Meyer, 1997), with benefits frequently asserted for repurposing and extending code. We will argue that most control techniques, including Isabelle tactics and IsaPlanner plans, are not very modular, so establishing modularity for Feasch models will indicate another difference. The specific hypothesis we will test is:

(EH4) Features contribute to modular design of control knowledge.

This is necessarily a subjective evaluation, primarily, so to attempt to address it scientifically will require specifying the evaluation criteria beforehand and applying this to case studies.

At the outset, let us note that in traditional approaches, the "module" is the tactic, plan, or ML function in its entirety. Steps in such a script cannot be used elsewhere — they must be copied and pasted, or else defined explicitly as a separate module. In Feasch, modules can be viewed at many levels: each individual feature, mapping, or method is autonomous and coherent; using hierarchical mappings, these can be grouped so that control knowledge for some domain acts as a higher-level module, organised so that it can function independently of other higher-level modules.

6.3.1. Meyer's Criteria

One common standard for judging modularity is that proposed by Meyer (1997). A *modular* computer program — or problem solving technique — will satisfy five criteria:

> **Decomposability**. A program is created by decomposing a problem into smaller subproblems solved separately.
>
> **Composability**. Modules can be freely combined to produce new systems.
>
> **Understandability**. Individual modules are understandable by a human reader .
>
> **Continuity**. A small change in the specification will result in changes to only a few modules, without affecting the architecture.
>
> **Protection**. An abnormal run-time condition will affect only a few modules, and the system will otherwise function robustly. (Meyer, 1997)

6.3.2. Evaluation

Let us look, in turn, at how each of these criteria stands up against the Feasch approach. Feasch requires decomposing control knowledge into individual features and specifying separately how these features trigger commands or other detectors. By recommending the use of detectors on the smallest level, it further encourages the decomposition of a complex decision procedure into features linked by mappings, as shown in the development of secondary "long-shot" control rules for HOL. By re-using the features for the initial detection of the relevant logical operators, these "long-shot" control rules also show composability applies to Feasch: features can be freely linked to other detectors or commands. Our generation of natural language explanations demonstrates that Feasch contributes to understandability, as does the observation of "useful failure" in our use of Feasch in automation. Protection was shown in how, when extensionality was lacking in the differentiation algorithm (§6.2.2), the technique cued the relevant method nevertheless, and, when it failed, it looked for other possible methods; not finding any, it returned control to the user who could repair the problem. Continuity was shown in the repair: no changes were needed to the control knowledge modules already created, and simply adding an additional feature and method sufficed to solve the problem.

This evaluation of our approach against standard modularity criteria appears to confirm (EH4). The subjective nature of the criteria preclude a quantitative evaluation, and whilst judgments of third parties would give stronger evidence, even such results would not "prove" this hypothesis. In keeping with the philosophy of science, what is crucial is that the hypothesis could be falsified: it is conceivable that our approach could have failed against the modularity criteria, whereas it is clear they did not.

If we look at other control knowledge techniques, we find that many of these modularity properties do not hold. For both tactics and plans, the control knowledge is expressed as one large entity which neither encourages the user to decompose the solution nor does it permit application or repurposing of pieces therein. The use of simplification rules demands decomposition, but there

is no alternative way of composing these elements special to such rules; specifically, when `simp` fails, the user is not given any usable feedback. The behaviour of `simp` in any instance can only be understood by reading an obscure trace, and the behaviour of most tactics is only understandable by looking at difficult-to-read source code. Likewise, on error, the only protection afforded to the system is if the tactic were not applied. IsaPlanner displays the modularity properties of understandability and protection, but for continuity, it — as with `simp` and tactics in general — is necessary to stop the current application, change the high-level module (at the source level), reload the relevant code, and re-apply. As we saw with the extensionality rule, control knowledge in Feasch can be added or modified at any point in the proof process. In all of these areas, it appears the Feasch supports modular expression of control knowledge to a greater or equal extent than each of the other techniques available in Isabelle.

6.4. Conclusions about the Expressivity of Features

Through the course of the last chapter and this, we have found four significant differences in how control knowledge is expressed in the current approach as compared with other techniques. The hypotheses we have tested are:

- **(EH1)** Features provide reasoned explanations for method selection, *e.g.* for use in generating natural language proofs with causal information.
- **(EH2)** Features enable useful guidance for interactive theorem proving.
- **(EH3)** Features can encode control knowledge for automated theorem proving.
- **(EH4)** Features contribute to modular design of control knowledge.

The first, second, and fourth were found to hold, and we observed that the corresponding statements were not generally true of most other techniques. The third hypothesis was confirmed for differentiation, but we observed that there would be problems using the current system for more complicated decision procedures; automated theorem proving is the thrust of other control knowledge techniques — exclusively, apart from IsaPlanner. It would be worthwhile to investigate enriching this capability for Feasch (§9.1.4.ii), but it is remarkable that it can be so useful with the present limited support in this area. We can conclude, on the basis of these expressivity properties being so different, that Feasch is a very different approach to encoding control knowledge, at least in Isabelle. Can these differences be used to show decisive benefits of this approach? Although some have been suggested in passing thus far, in the next section we will turn our focus specifically to this question.

Part Three: The Benefits of Features

Chapter 7. Semi-Automated Theorem Proving: Integral Calculus

Feasch, we have observed, can be used for both interactive and automated theorem proving, and furthermore it offers inspectability and modularity beyond the other techniques available in Isabelle. For these reasons, we could expect Feasch to have particular benefits for problem solving in mixed environments, where one or more machine agents and a human user can all contribute to the problem solving process. This leads to a first benefit hypothesis:

> **(BH1)** The inspectability and modularity properties of feature-based control knowledge are beneficial to multi-agent theorem proving.

We will look specifically at complex problem solving, where perfect control knowledge, such as decision procedures, cannot be found. An inspectable technique will permit other agents, *e.g.*, a human, to use partial results of problem solving. A modular technique will permit developers to extend or modify the pre-existing control knowledge to make it more appropriate for domains of interest, and will permit other agents to focus on particular modules which seem appropriate.

7.1. Background to the Problem

The problem we chose to investigate for this hypothesis is indefinite integration. Integrals are usually solved by using a small number of methods; the choice of methods, however, and the choice of arguments used, can vary widely. Mathematicians are usually very good at choosing an appropriate method, but the process behind this remains obscure: it is difficult to teach, it is typically acquired by much experience, and it is very different to how machine systems work.

7.1.1. History: SAINT, SIN, and a Decision Procedure

Slagle's program SAINT, the Symbolic Automatic INTegrator, is the first and most straightforward computer program which evaluates integrals, achieving "the level of a good college freshman" — a very good level, in fact, solving 52 of 54 problems on the final exam set for the first year course at MIT (1963). It works by using 26 "standard forms" for which the solution is immediately recognised, 8 transformation rules (including linearity and linear arguments) and 10 heuristic transformations (including trigonometric identity replacement and various types of substitution). The transformations are applied like production rules, with the transformation rules taking priority, and with preconditions for each determining whether the integrand is "amenable". A goal list keeps track of the different solution paths the program is

considering, and a cost estimation selects the most promising path at any time. Later programs, including SIN (Moses, 1967) and Macsyma, have extended this type of automated integral solving with better algebra handling routines, sophisticated cost estimations, and a number of more advanced solution methods.

After 1969, however, interest in automated integrators such as these dropped off, after the discovery by Risch that integrals can be solved theoretically using Louiville's theorem and algebraic field extensions. Louiville's theorem, in modern language, says that if an integral can be evaluated, as an operator from elementary functions to elementary functions, then the answer will be a sum of one part in the original field and possibly some parts which are logarithmic over the original field extended by some algebraic constants (Risch, 1969; Neubacher, 1992). The Risch algorithm and the later, faster Risch-Norman algorithm use this "structure theorem" as the basis for a method guaranteed to solve any solvable integral, by trying all the possible expressions (a large, but finite number) which could be the integral of the presented problem.

Some modern computer algebra systems use an implementation of this algorithm as the basis of their integration routines. The algorithm is very intricate, programmatically, however, and many CAS programs rely on a large number of recognised solutions (a lookup table), with a small number of methods for converting a presented problem to one of these forms. Where the Risch algorithm is used, the systems are insensitive to boundary conditions and occasionally return false statements, such as the following:

$$\int x^{n-1}\,dx = \frac{x^n}{n} + C$$

The statement above is true so long as $n \neq 0$, but a correct solution must include the special case that $\int x^{-1}\,dx = \ln x + C$. A program would fail if it relied on the general solution and instantiated it where $n = 0$; while mathematicians will recognise this, errors such as this could be catastrophic if the tools are used in a mission-critical engineering application.

For its ability to reason with due consideration to boundary conditions, Isabelle will offer some advantages as an environment for solving integrals. We note that these boundary conditions have also been ignored in SAINT and other work outwith the Risch tradition; for this reason, the current investigation could yield a useful applied system. Additionally, we note that many expressions which are not integrable can be "partially integrated" to the point where the integral can be expressed in terms of known "special functions". The Risch and Risch-Norman algorithms do not apply in these cases, and although we are not specifically addressing this problem, it seems likely that the present work could have benefits in this area.

7.1.2. Methods for Human Integral-Solving

The first step in constructing a system for solving integrals which can function interactively with a human agent is to identify the methods commonly used. We will base this analysis on the first-year calculus course taught at Edinburgh University. Firstly, the integrals to several functions are taught explicitly: $x^n (n \neq 1)$, $1/x$, $\ln x$, e^x, the six trigonometric functions, the three principal hyperbolic functions, and five functions with quadratic denominators $\frac{1}{a^2+x^2}$, $\frac{1}{a^2-x^2}$, $\frac{1}{\sqrt{a^2-x^2}}$, $\frac{1}{\sqrt{a^2+x^2}}$, and $\frac{1}{\sqrt{x^2-a^2}}$. These are listed, for example, on a one-page summary study sheet for the course in calculus (Arthur, 2000) and in key tables in many textbooks and web sites. Methods for solving these integrals should be included in our system. Additionally, the following common integration rules are listed in the summary study sheet:

$$\int \left(a \cdot f(x) + b \cdot g(x)\right) dx = a \int f(x)\, dx + b \int g(x)\, dx \qquad \text{(linearity)}$$

$$\int_0^x f^{-1}(t)\, dt = x \cdot f^{-1}(x) - \int_{f^{-1}(0)}^{f^{-1}(x)} f(t)\, dt \qquad \text{(inverse function rule)}$$

$$\int u'(x)\, v(x)\, dx = u(x)\, v(x) - \int u(x)\, v'(x)\, dx \qquad \text{(integration by parts)}$$

$$\int f(ax+b)\, dx = \frac{1}{a} F(ax+b) + C \qquad \text{(linear arguments)}$$

$$\int f(u(x)) \frac{du}{dx}\, dx = F(u) + C \qquad \text{(reverse chain rule)}$$

(Where F appears it is understood that it is an antiderivative of f, that is $F'(x) = f(x)$.) The inverse function rule is scarcely taught in the course and is not given in its indefinite form ($\int f^{-1}(x)\, dx = x \cdot f^{-1}(x) - \left[\int f(u)\, du\right]_{u=f^{-1}(x)}$), although inverse substitution (a type of u-substitution below) gives the equivalent result. Other courses may treat these slightly differently, skipping the inverse function rule, splitting linearity into two rules (addition and constant multiplication), or calling the reverse chain rule as "derivative-divides", but this group of basic rules is standard.

The course also teaches a number of techniques which do not rely on these integration rules but rather on the clever use of concepts students have already learned. The first group of these involves the replacement of an expression in the integrand with one that is equivalent. Algebraic manipulation is the simplest type of replacement, and students learn early in the course that they can expand, factor, or otherwise simplify algebraic expressions in an integrand. For integrating rational functions (polynomial fractions), students learn the technique of partial fractions: $\frac{A(x)}{P(x)Q(x)}$ is replaced by a sum of smaller polynomial fractions $\frac{A_1(x)}{P(x)} + \frac{A_2(x)}{Q(x)}$, each of which is hopefully easier to integrate. Manipulating the expression with trigonometric identities (*e.g.*, $\sin^2 x = 1 - \cos^2 x$ and $\sin x \cdot \cos x = (\sin 2x)/2$) is another replacement method.

Substitution is the general technique of introducing an expression in a new variable equal to an expression in the integrand, and then changing the integrand to be purely in the new variable of integration. The integral is then solved, and converted back to the original variable of integration. The simplest form of this is u-substitution, where u substitutes for some expression in

the integral (say, in terms of x), and dx is replaced by an expression in du: for instance, in solving $\int (x+1)^2\,dx$ one could set $u = x+1$, compute $du = dx$, evaluate $\int u^2\,du = \frac{u^3}{3} + C$, then convert back to get $\frac{(x+1)^3}{3} + C$. (This is a derivation for the linear transformation rule.) u-substitution is taught as particularly useful with inverse functions (for $\int \sin^{-1} x\,dx$, let $u = \sin^{-1} x$ so $x = \sin u$, giving $\int u \cdot \cos u\,du$) and "nasty terms" (u replaces an unpleasant expression in the integrand). Trigonometric substitution is the introduction of a trigonometric expression for the variable of integration (such as $x = \sin u$), to simplify such an expression as $\sqrt{1-x^2}$). Tangent substitution is used to change a large polynomial fraction in trigonometric functions to a polynomial fraction in a variable. These substitution techniques are taught explicitly in the course, with conditions when they are normally appropriate. A final form taught in the course — the most general form of substitution — is "implicit substitution", when some expression $g(u)$ substitutes for an expression $f(x)$ occurring in the integral. (Tangent substitution is sometimes omitted from other calculus courses.)

There are a number of other advanced techniques for solving integrals. Some can be described as replacement or substitution, but as they are distinctly treated in the Edinburgh course, we identify them here. (They are quite difficult, however, and we do not expect to encounter them very frequently. They are sometimes present in other calculus courses and sometimes omitted; likewise, there may be one or two specialised techniques not in our focus course.) The techniques taught at Edinburgh are: expanding power series, solving recursively, and integrating quadratic denominators. (A fourth technique, reduction, applies only to definite integrals and so is not discussed here.) With power series, expressions in the integrand are replaced by their Taylor or Maclaurin expansion, combined algebraically (multiplication or composition) or split using linearity, and then converted back to a recognised combination of functions, where possible. Solving an integral recursively means that by some method (often integration by parts), an expression is derived relating the integral to a function of itself, *e.g.*, $\int f(x)\,dx = g(x) + k \cdot \int f(x)\,dx$; in some cases, this expression can be rearranged to give a solution for the integral ($\int f(x)\,dx = \frac{g(x)}{1-k}$). The technique for integrating expressions with quadratic denominators involves completing the square (an algebraic manipulation) to get the integrand into a recognised form (which might be the derivative of $\tan^{-1} x$).

No list of methods for mathematics can ever be exhaustive, but this list is very similar to those in college textbooks and to the techniques which we learned at school. To summarise, the list of methods is given below, noting an abbreviation for it that will be used as its Feasch name:

- recognise solution (solve)
- linearity rule (linearity)
- inverse function rule (inv rule)
- integration by parts (by parts)
- linear arguments rule (linear args)
- general/reverse chain rule (chain)
- factor algebra (factor)
- expand algebra (expand)
- simplify algebra (simpl)
- partial fractions replacement (partial fracs)
- trigonometric identity replacement (trig id)
- substitution—u (u-subst)
- substitution—trigonometric (trig-subst)
- substitution—tangent (tan-subst)
- substitution—implicit (other subst)
- power series (pow series)
- recursive (recursive)
- quadratic denominators (quad den)

7.1.3. Features for Human Integral-Solving

In contrast to these methods, where there is a general consensus in teaching, the reasons people choose methods is very poorly understood. We performed an exploratory experiment with 10 subjects where they were asked to solve integrals using the speak-aloud protocol. Subjects were asked to explain why they chose to make any step (efficient cause, *cf.* §5.2), in order for us to gain an understanding about any features they may be using. Some qualitative results — from subjects as well as the course notes, textbooks, and other experience — are summarised as follows:

> "If you've got a lot of messy algebra, try u-substitution."
>
> "Integration by parts works well when there's a product of x or something easily differentiable with something integrable."
>
> "When there's an ugly fraction of polynomials, try partial fractions and then the big gun, tangent substitutions for quadratic denominators."

Parameter Instantiation

Some of the methods require the selection of one or two parameters: for example, u-substitution is seeded with the expression which u substitutes for, and integration by parts needs to know which parts to split the integrand into. How people choose these parameters will require close study, because unlike the small set of methods, there are an infinite number of possible parameters. This question gives a good test of the two approaches because they explain this selection process very differently. Example suggestions for choosing these parameters are:

> "Use trig substitution to replace an x^2 term with $\tan^2 u$."
>
> "If the integrand has a simple derivative, try a trivial integration by parts using 1 as the part to integrate and the integrand as the differentiation part."

Mappings

We have also, on the basis of these experiments and textbooks, identified the methods chosen when certain features are detected. The list of features and the methods they cue is shown in figure 7.1. Neither the list nor the mapping is conclusive, but it is an approximate basis on which to create a Feasch model of control knowledge, and we can reasonably expect the features and the methods to be understandable to users of our systems.

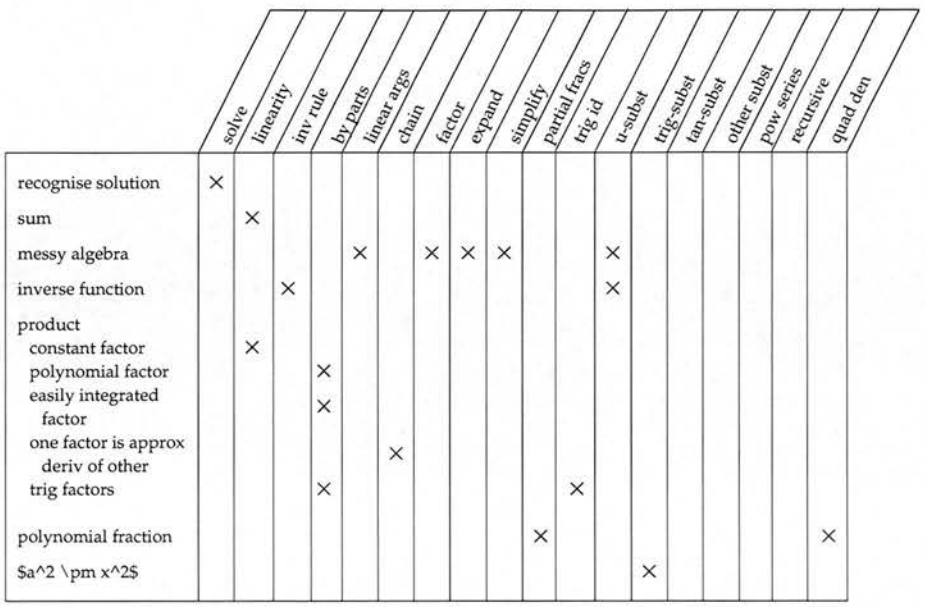

FIGURE 7.1: **Features for Integration Methods**. From analysing human subjects, a number of features were identified which seem to trigger the choice of method for solving an integral.

7.2. A Feature Wizard for Integration

The majority of these methods have been encoded as rules in our Isabelle theory of calculus. Detectors for each feature have been defined, and, as with differentiation, their execution is cued by a single high-level detector of `IntegralE0`. Mappings follow the matrix shown in figure 7.1.

The methods not implemented involve algebraic manipulation. This is a difficult problem for which many other techniques exist, and we will assume that another agent will be able to handle these problems, *e.g.*, the built-in tactic `simp`, a CA system such as Maple, or the human user. Features are used to note when such methods should be applied.

7.2.1. GUI: Crossing the "Auto-Run Threshold"

So far we have seen the "point-and-click" interface for interactive theorem proving and the "fast-forward" automation technique. The "Settings" tab allows a user to configure the behaviour of the wizard, including the option to enable a third aspect of the Feature Wizard graphical user interface, the "auto-run threshold". If this is turned on, the system will automatically try to apply a command if its cue exceeds the specified threshold.

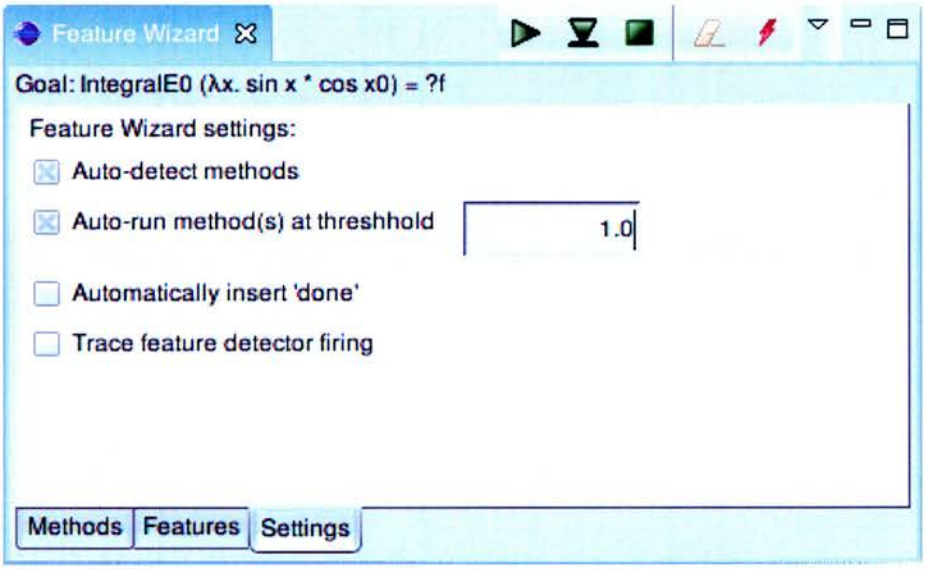

FIGURE 7.2: **Feature Wizard Settings.** The behaviour of the Feature Wizard is customisable in the "Settings" tab. One setting, the "auto-run threshold", is particularly useful for semi-automated theorem proving. Any methods cued above this threshold will be automatically inserted into the proof in an otherwise interactive context.

We will use this option to explore the use of Feasch with Isabelle for semi-automated theorem proving. When defining our mappings, we used the convention that a cue greater than or equal to 1.0 means that a method is good or safe to apply; by specifying this as the "auto-run" threshold, Feasch will run these methods automatically. If these do not apply, or if there are none so strongly triggered, the user will be presented with the lists of methods and features and can proceed interactively.

7.2.2. Evaluation

Our control knowledge is able to solve many problems automatically, including $\int (\sin x)(\cos x)\,\mathrm{d}x$ and $\int e^{2x-1}\,\mathrm{d}x$. The biggest restriction, as in the case of differentiation in the last chapter, is the ability of the system to reduce algebraic expressions. In the problem of $\int x \ln x\,\mathrm{d}x$, the system suggests integration by parts and is able to reduce the expression to $(\ln x) \cdot \frac{x^2}{2} - \int \frac{1}{x} \cdot \frac{x^2}{2}\,\mathrm{d}x$, but gets stuck on the "easy" part of simplifying the resulting integrand to be $\frac{x}{2}$, which it would also be able to integrate. Similarly, in the case of $\int \sqrt{1-x^2}\,\mathrm{d}x$, it can find the appropriate trigonometric substitution ($x = \sin u$) and even evaluate the square root using `simp`, but it does not know to reduce the resulting expression $\int (\sin x)(\sin x)\,\mathrm{d}x$ to $\int (\sin^2 x)\,\mathrm{d}x$ or how to rearrange the identity $\cos 2x = 1 - 2\sin^2 x$ so that it could be substituted.

Importantly, for the problems that our system is unable to solve, it permits the participation of a human agent — the user. A success of the Feasch approach — and a benefit for semi-automated theorem proving — is that it provides suggestions to the user when none of the methods are ranked with a strong weight. For instantiations, one of the most difficult parts of the many techniques, the user can choose the rule to apply from the ranked list and manually type only the instantiation, *e.g.*, the u to substitute.

Because the features are explicitly presented, a third agent could also be involved in the process: of particular interest would be to invoke a computer algebra system to assist with the weakness of the present system in manipulating algebra. This could, for example, analyse the suitability of various candidates for u-substitution (u expressed as a function of x) by computing $\mathrm{d}x$ in terms of $\mathrm{d}u$ and reducing the resulting product $u\,\mathrm{d}u$. It would be interesting to explore this prospect, but here we note that such collaborative problem solving is greatly enhanced by the inspectability of our approach.

By altering the weights in our control knowledge, the influence of certain features can be adjusted, with the result that the system may try techniques in different orders depending on the problem. Users can use this capability to vary the behaviour of the system to match their preferences. One user, for example, might prefer u-substitution:

$$\begin{array}{lll} \int(\sin x)(\cos x)\,\mathrm{d}x & = \int u\,\mathrm{d}u & \text{u-subst, } u = \sin x \\ & = \frac{u^2}{2} + C & \text{solve} \\ & = \frac{\sin^2 x}{2} + C & \text{simpl (manual/external)} \end{array}$$

Another user might set the product of two expressions in the integrand to be a strong cue for integration by parts:

$$\begin{array}{lll} \int(\sin x)(\cos x)\,\mathrm{d}x & = \sin^2 x - \int(\cos x)(\sin x)\,\mathrm{d}x & \text{by parts} \\ 2\int(\sin x)(\cos x)\,\mathrm{d}x & = \sin^2 x + C & \text{simpl (manual/external)} \\ \int(\sin x)(\cos x)\,\mathrm{d}x & = \frac{\sin^2 x}{2} + C & \text{simpl (manual/external)} \end{array}$$

Yet another user may have defined control knowledge to recognise trigonometric identities, with a strong cue sent to reduce expressions, if possible, when they occur inside an integrand:

$$\begin{array}{lll} \int(\sin x)(\cos x)\,\mathrm{d}x & = \int \frac{(\sin 2x)}{2}\,\mathrm{d}x & \text{trig-id} \\ & = \frac{1}{2}\int(\sin 2x)\,\mathrm{d}x & \text{linearity} \\ & = \frac{1}{2}\int(\sin u)\,\frac{\mathrm{d}u}{2} & \text{u-subst, } u = 2x \\ & = \frac{1}{4}\int(\sin u)\,\mathrm{d}u & \text{linearity} \\ & = \frac{1}{4}\cos u & \text{solve} \\ & = \frac{\cos 2x}{4} & \text{simpl (manual/external)} \end{array}$$

7.2.3. Conclusions

Our approach, like many of the others, has been able to solve some very difficult problems, fully automatically. We have been unable to get access to SAINT, SIN, or Macsyma, or to find the actual problems used in their evaluation, so it is not possible to give a quantitative comparison. However, our emphasis has been slightly different and, as we have not developed significant algebra facilities, it is likely that our system alone operates at a lower level of proficiency. There are, however, three important distinctions to our system. Firstly, it is able to reason with boundary conclusions, automatically discharging *e.g.*, many proofs of integrability or continuity requirements; these are typically ignored in the integral-solving systems we have seen, and we doubt that they are considered in SAINT and SIN (although we have not been able to find out for

sure). Whilst we have not shown this specifically for integration, the behaviour of the system on such problems, *e.g.*, $\int \ln x \, dx$, is very similar to that described in §6.2.3 for differentiation.

Secondly, a user can engage with our system interactively when automation fails or is not desired. Furthermore, it is not difficult to see how an additional agent — such as a computer algebra system — could be invoked to apply a method, to simplify or factor algebra (producing a subgoal for the equality of the simplified term, which may be easier to solve than the simplification alone), or even to contribute features which are more easily detected in that environment. Because these collaborative capabilities are not available in the other integral-solving systems, we believe that students working with Feasch, for example, would outperform a team consisting of a system such as SAINT; we would have liked to conduct this and other experiments to provide a formal comparison, but, as has been noted, these other systems are not available.

Finally, our system is able to communicate why methods are suggested as appropriate. This has the benefit that a user can easily understand the control knowledge we have provided; she can follow or decline the suggestions made by the Feature Wizard, but by knowing the reasons the suggestions were made, she is in a much better position for making this choice. For all these reasons, we conclude that (BH1) is shown for our system: the inspectability and modularity properties of feature-based control knowledge are beneficial to multi-agent theorem proving.

Chapter 8. Learning with Features

An interesting property of the Feasch approach is the introduction of an abstract layer — the features — between the problem representation and the chosen methods. This division, we suggest, could have benefits for learning part (or perhaps even all) of the control knowledge automatically, reducing the amount of effort needed by a developer. Features could potentially be learned by looking for common patterns in proof states, and methods, likewise could be learned by looking for common patterns in proof steps. The mappings between these could be adjusted depending on which features were found when the various methods were applied. To test this supposition, we set up our second benefit hypothesis:

> **(BH2)** Feasch allows the use of learning mechanisms not applicable to other systems, simplifying the encoding of control knowledge and yielding improved performance.

We will look specifically at the ability to learn mappings automatically from a corpus of proofs in the domain of differentiation, using the features and techniques from Chapter 6. Ideas for learning features and methods are discussed as future work (§9.1.5).

8.1. Learning Mappings from Solved Derivatives

The process we will use to attempt to learn mappings is to provide the feature detectors and command schemas of interest and then to analyse a selection of sample proofs. By comparing the presence of each feature against the commands applied at every step, we establish a relevance vector for each ⟨feature,schema⟩ pair. These vectors are used to generate mappings which are added to the provided control knowledge to form the test model. The test model's performance is then evaluated on another, distinct set of sample problems.

Looking at the problem of solving integrals, we have a set of control knowledge from Chapter 6. Writing the mappings — and choosing the weights — was one of the most mundane parts of specifying this control; if Feasch is able to do this automatically, the developer's job is dramatically simplified. In the domain of integration, there is an idealised goal of optimal performance, *viz.*, whether the decision procedure is learned. This achievement would conclusively demonstrate the hypothesis and provide strong support for using features to express control knowledge.

8.1.1. Training Methodology

Our development of the calculus theory in Isabelle has yielded 19 features and 21 command schemas useful for differentiation. We constructed terms to differentiate by taking random combinations of the functions in our theory — addition, subtraction, negation, multiplication, division, multiplicative inverse, exponentiation ($\mathbb{N}$), exponentiation ($\mathbb{R}$), sine, cosine, exponentiation (e), and the natural logarithm — with arguments chosen with equal probability as either our functional variable (x) or a randomly selected constant, up to size of 10 (in Isabelle's definition, corresponding number or operators and elementary terms). We then applied our decision procedure to take the derivative of each such term to generate a training set of 100 sample proofs. We deferred all subgoals which could not be proved and, as noted in §6.2.3.iii, found solutions to all 100 problems. The total number of non-deferring steps was 1117.

The applicability of each feature $\mathcal{F}$ was recorded by counting, for each distinct operator schema σ, three situations:

> **success_count**$_{\mathcal{F},\sigma}$: the number of steps in proofs in the training pool where σ was applied, $\mathcal{F}$ was detected in the state immediately prior, and all instantiations required for σ in this step were contained in $\mathcal{F}$?
>
> **incomplete_count**$_{\mathcal{F},\sigma}$: the number of steps in proofs in the training pool where σ was applied, $\mathcal{F}$ was detected in the state immediately prior, but at least one of the instantiations required for σ in this step was not contained in $\mathcal{F}$
>
> **fail_count**$_{\mathcal{F},\sigma}$: the number of steps in proofs in the training pool where σ was applied and $\mathcal{F}$ was not present in the state immediately prior

At the end of training, this gives a vector [success_count,incomplete_count,fail_count] for each pair [feature $\mathcal{F}$,operator σ]. We set the weight of the mapping from $\mathcal{F}$ to σ as

$$w_{\mathcal{F},\sigma} := \frac{0.01 + 10 \cdot \text{success\_count} + \text{incomplete\_count}}{1 + 10 \cdot \text{success\_count} + 5 \cdot \text{incomplete\_count} + 3 \cdot \text{fail\_count}}$$

This formula was chosen from experience with neural network and statistical learning paradigms, as a rough mechanism for discouraging incorrect cues without eliminating correct cues, and spreading the credit when multiple features cue a schema. The parameters have not been tuned, although the potential for this is discussed in §8.2.

As an example, consider an operator schema that is applied a total of twelve times in our training pool. If a particular feature is present once with enough information to instantiate the operator schema fully and thrice with incomplete information, our vector is [success_count=1,incomplete_count=3,fail_count=8], and the mapping from this feature to this operator schema is computed to be $(0.01 + 10 + 3)/(1 + 10 + 5 \cdot 3 + 3 \cdot 8) = 13.01/50 \approx 0.26$.

We then set initial mapping $w_{0,\mathcal{F}}$ on a feature $\mathcal{F}$ using the same computation as for $w_{\mathcal{F},\sigma}$ on the vector $[\sum_\sigma \text{success\_count}_{\mathcal{F},\sigma}, \sum_\sigma \text{incomplete\_count}_{\mathcal{F},\sigma}, \sum_\sigma \text{fail\_count}_{\mathcal{F},\sigma}]$.

8.1.2. Experimentation

This process yields a set of mappings between the two control knowledge elements — features and methods — provided. These three elements together give a Feasch model which can be used through the Feature Wizard, semi-automatically or interactively, as discussed in chapters 4-8. For the present purposes, we will evaluate the quality of the learning by using the automated problem solver discussed in §6.2.3.iii, replacing the hand-coded mappings there by these the resulting mappings. We will run the new control knowledge on a distinct set of 100 randomly generated problems and assess the performance on three levels:

- Number of problems solved
- Percentage of these problems with assumptions generated
- Total number of steps
- Total number of method attempts

To contextualise these results, we will compare against these results against the decision procedure manually created in Chapter 6.

8.1.3. Results and Analysis

The resulting Feasch model using mappings learned in this way was able to solve all the problems in the test case. The percentage of problems where assumptions were introduced and the total number of steps were both approximately the same as for the hand-coded model — within the margin of error — which would be expected as these numbers are primarily dependent on the problem itself. The secondary key performance indicator is the number of mis-steps made: here, the learned mappings were significantly worse than the hand-coded model, applying 1475 inappropriate steps compared with 762. It is likely that this could be improved either by adjusting the learning algorithm or providing more training examples, and it would be interesting to discover how close to perfect selection can be achieved. Although the difference is significant, it does not represent a dramatic deviation in performance: the models are boh on the same order of magnitude, with one wrong step on average tried before the right step. Trying one wrong method is not expensive and, compared with our own experience at proving theorems in Isabelle, is quite a good result.

	Learned Mappings	**Hand-coded Mappings**
Number Solved	100	100
Percentage with Assumptions	44%	39%
Total Steps	1160	1117
Steps Attempted	2625	1879

FIGURE 8.1: Performance of Learned Mappings

Based on these results, we conclude that the system is able to learn appropriate mappings, given a large enough set of training data. The performance is slightly worse than our hand-coded mappings, in this case, but it is surprisingly close. For sure, it is a great improvement over the default case where no mappings are provided! This confirms (BH2).

8.2. Discussion

This performance, we note, could likely be improved by tuning the learning algorithm. The parameters were chosen as simple estimates and the factors as simple counts; in both areas there is much room for improvement. More complicated techniques could also be applied in the post-processing. In particular, we would be interested in using principal component analysis to identify the few features most relevant to a method's selection, and then restricting the mappings generated to those which appear to have a significant influence. We also note that there is often only limited success which can be achieved merely by analysing data. In this experiment, for example, there were frequently two or more methods which could have been applied, but our analysis only considered a single application. A better approach would be to enable the system to test hypotheses about relationships between features and methods, updating the mappings in real-time as the system tries to apply the candidates it has cued.

Despite these criticisms, this experiment establishes that the mappings do not always need to be provided. The decision procedure for differentiation was learned merely by specifying what commands are used and what features might be important, and by providing sample solutions. Other approaches to learning in automated theorem proving have not, to our knowledge, achieved such results, and we attribute the result here to the introduction of explicit features which enrich the problem description and focus attention on the key aspects of the problem. When writing control knowledge for a new domain, it will be much easier for a developer to describe only potential features and ways of applying methods, without having to consider the relationship between them. Furthermore, this could permit other types of learning to be conducted with a more narrow focus, *viz.*, finding new features and finding new methods, and as in the case of the human developer, these machine algorithms also need not consider the relationship between them. Some ideas for learning these other components — and for testing them experimentally — are described in §9.1.5. Here, we have shown that the difficult task of learning control knowledge is rendered easier by dividing the space of what needs to be learned. Using features as an intermediate layer, between problem representation and problem solution, in addition to the benefits for inspectable, semi-automatable theorem proving, allow part of the solution process to be learned automatically.

Part Four: The Future of Features

Chapter 9. Critical Analysis and Future Work

Our experiments have shown encouraging results for the Feasch approach, but more importantly, they have raised a number of interesting research issues. With regards to Feasch with Isabelle, we make specific criticisms in several areas: showing and improving usability, a focus on syntactic features, the numeric mappings, planning capabilities, and prospects for learning. In each of these areas, we have ideas for how future work can address the limitations discovered thus far. More generally, we have focussed in this thesis on one application of the approach, that of semi-automated theorem proving; we are very keen to see how it would fare in other areas, both for computational problem solving and for cognitive modelling.

9.1. Isabelle with Feasch

An extension to this work which could be very useful is to extending the coverage of Isabelle theories by Feasch control knowledge. Some of the difficulties we have encountered suggest areas where this could be particularly useful, *e.g.*, algebraic simplification (Chapter 6), because the current techniques in Isabelle sometimes leave much to be desired: in different situations, different types of simplifications may be appropriate, and this could be cued by Feasch or adjusted by a user; in some instances, *e.g.*, factorisation, it could be possible to use an external system to suggest features which may cue certain solution strategies. The thrust of our research, however, is on theoretical aspects of problem solving, and in this light we are especially interested in using these areas in ways could further understanding and resolution of the limitations of the current work.

9.1.1. Usability Analysis

We have found the Feature Wizard helpful, as have a number of other people in informal settings, and we believe we have demonstrated in the course of this thesis why it is likely to be so. However, the argument for its usability could be strengthened significantly by undertaking some specific quantitative analyses. We would like to perform experiments with human subjects in three areas: interactive theorem proving, semi-automated theorem proving, and developing control knowledge. The first two areas could provide subjective data about user's experiences, including whether they found the system helpful, as well as objective information such as whether

they were any faster at achieving goals than a control group using Isabelle without Feasch. Not only would this provide additional evidence about the usefulness of our system, it could also give us feedback about which aspects of the system are most useful and which could benefit from attention.

The third area of usability experiments would be most interesting, as one of the primary goals has been to provide a framework for expressing theorem proving control knowledge, but it would also likely be the most work. The specific questions we would be interested in here are whether users find control knowledge expressed in FDL understandable, whether developers can express their own control knowledge in FDL, and how well FDL control knowledge can be extended by other users. Comparisons to other techniques would provide useful evidence about whether our system can serve as a useful tool, and insight into the actual difficulties users have — particularly with respect to extending other code — would be valuable to us in improving the feature detection language.

We also have several ideas for how the current user interface could be improved to make the entire system more usable. The first of these is to attempt to relate entities in the proof state to the features and methods which are related to it, either using a visual cue such as colour or through interactive feedback such as on mouse hover events. In an extension to this, it could be possible to bring up cued methods in response to a right-click on a related portion of the current subgoal: for example, if a user clicks on a term in some assumption, they might see a menu showing methods which use that term, either using the entire assumption in a forwards-rule application or cued by features which identify that term. Another idea is to add support for hierarchical proof representations, so that minor steps can be hidden by a user wanting a high-level view of the proof; this hierarchy could be suggested by features and methods, and natural language comments about the reasons for the method selection could be included. Further potential enhancements include being able to turn on and off predefined groups of feature detectors (called "tools", and some support for this is in the code, though not the UI), displaying a graph of the causal hierarchy of a particular method and of the current execution hierarchy, and easily being able to edit the mappings and weights (*e.g.*, using such a graph). Finally, it would be useful to have a visualisation of multiple simultaneous proof attempts in Isabelle (*cf.* §9.1.4)[33].

9.1.2. Non-Syntactic Features

The features used in the experiments in Chapters 5 through 8 have primarily involved syntactic abstractions of portions of the goal. Features used in human problem solving almost certainly have a much broader range, and it would be interesting to explore the prospect of using more varied features in an artificial context.

Geometry is one area where this is an intriguing possibility. Various programs exist for drawing diagrams for proof states (Wilson & Fleuriot, in press; Winterstein, 2004). Detectors which analyse this diagram could test for features not apparent in the syntactic problem representation, and, in a suitable domain, it seems to us that such features might be useful for cueing schemas which would not otherwise have been found. Interestingly, this appears very close to how humans reason (*ibid.*).

[33] We created a component for this which worked with λClam, but we have not yet been able to port it due to the challenges of representing a plan space in Isabelle. We are hopeful that this will be facilitated by IsaPlanner.

9.1.3. Issues with Mappings: Numeric Weights and the Compositionality of Features

Turning our attention from the features to the mappings, we find two aspects of the current approach to cueing schemas unsatisfactory. The use of numeric weights is unattractive for many reasons, and there is frequently redundancy in the feature detectors used in Feasch with Isabelle.

The Ugliness of Numbers

Although the architecture in Chapter 3 allows an arbitrary implementation of `IMapping`, the classes we have presented all use a numeric weight. (The simplest, used in `ListExecutor`, is equivalent to a universal weight of 1, so we include it in this critique.) While this technique is commonly used, in techniques from evaluation functions to neural networks, it could be a source of irritation amongst users. Weights in different domains could be difficult to compare, and the resulting lack of consistency, we expect, would lead to difficulties in composing different schemas. The resulting lack of modularity could be a significant challenge in a large system.

We have introduced some conventions, such as to use 1.0 as a divide between methods which have some guarantee of appropriateness and those which do not, but we feel these conventions do not go far enough. The prospect of inferring weights automatically (Chapter 8) offers some assistance to the problem, but it is necessarily sensitive to training data and such data is not always available. Unfortunately, we do not at present have any alternatives which would be better.

Re-using Featural Information

A second limitation of the current mapping strategy, in practice, is that it is difficult to use information provided by features in subsequent feature detectors. While the proof command schemas can use the instantiated arguments of the features which cue it, feature detectors in Feasch with Isabelle cannot. We note that this is not a problem in the underlying architecture, as the semantics for a cue to a feature detector include this information, but this information is ignored in the ML code which invokes the feature detectors and no support for accessing it is afforded in FDL. As a result, subsequent detectors frequently re-compute the arguments which would have been provided by the featural cue, as in:

```
feature "the conclusion is a conjunction" [] (the conclusion is "?P & ?Q");
feature "the left-hand conjunct of the
    conclusion matches an assumption" ["P","Q"]
    ((the conclusion is "?P & ?Q") AND (some assumption is "?P"));
```

Fixing this would not be a major change to the code, and we expect to be able to complete it in the near future.

9.1.4. Planning by Continuation: Persistent Features

Statefulness and Search

Wee see the most serious restriction of the current system as the lack of built-in support for search and planning. These techniques have been instrumental in the success of AI, and while it is remarkable that Feasch has been useful without it, it could be far more useful if it had these capabilities for high-level executive control (§6.2.3.v). If these capabilities are incompatible with the approach, furthermore, we have a catastrophic challenge to its use either as a theory of human problem solving or as an exclusive means of specifying control knowledge.

The problem can be viewed as one of statelessness: at every step in the theorem proving process, Feasch looks at the problem state from first principles, with no knowledge of its history. An intelligent solver will remember what it has been doing, and in either a plan or a search script, it will also know where the last step fits in the context of the overall strategy. There is no reason why this information could not be represented using features, and there is only one aspect of the approach thus far which would need to be changed. Rather than clearing the features at each new step, the system should support "persistent features" which are carried over, possibly with some changes. Tactics and plans could thus be implemented by carrying a feature such as "we are applying plan step A in proof state N" from one state N into the next state $N+1$; if a new feature "proof state $N+1$" is added to the context of new proof states, a "continuation mapping" could map the combination of these features to cue "do plan step B".

There are arguments that could be made for encoding tactics in this way, such as greater flexibility and interruptibility, but we recognise that in its current form it is far from elegant It would be worthwhile to explore, but with a view to refining it to a point where the use of features is natural for tactics and not, as it seems here, a laborious introduction of labels. (In §10.1.3.i, in the next chapter, we reason quite abstractly to come up with another solution which seems preferable.)

Using IsaPlanner with Feasch

One alternative to the use of continuations is suggested by the success of Feasch with Isabelle in a collaborative context, as demonstrated by showing hypothesis (BH1). It would be possible to have agents for planning or tactics analysing the problem state alongside Feasch, with these agents keeping track of their notion of context and suggesting the next steps through features and method cues to Feasch. IsaPlanner has particularly good support for exploring a space of proof states, through extensive low-level ML libraries with efficient caching of states and of individual assumptions. It also (as its name suggests) allows for specifying plans using tacticals and permitting gaps. These are all areas where Feasch with Isabelle is lacking. If the two approaches to control knowledge were interoperable, users would be able to express both tactical and heuristic ideas.

One interesting aspect of such a combination is that features could be used at the meta-level to encode the applicability of plans; a major weakness of plans and tactics in Isabelle currently is that there has been no way to do this, but a good integration of our approach with these approaches could yield that. Furthermore, we would expect many of the same benefits that were observed at the proof step level to be observed at the plan level: that the reasons for selecting plans are inspectable and extensible, and that some parts of the control knowledge for plans could be learned automatically.

9.1.5. Learning from Experience and Analogy

Learning Mappings On-line

The last chapter illustrated how mappings can be learned by a static analysis on completed proofs. An extension of this work could see Feasch update mappings based on what is selected in real-time and on what is observed to succeed. This would have the advantage the proof corpora are not required, and the system could adjust in response to what a user thinks is appropriate. Such a system would have the attractive problem that, over time, its suggestions become more closely aligned with a particular user and to the types of problems he solves.

Alternatively, the learning system could run unsupervised. When unattended, it could automatically apply the methods its control knowledge suggests and then update the mappings depending on how successful a method turns out to be. This could be interesting in either of the other mathematical domains we have explored, solving integrals or discovering when the basic HOL methods are appropriate.

Learning Features

In many instances, particularly in HOL, features are obvious to us from the shape of a rule. This "obviousness" can be formally described as the conditions for a rule's applicability, and these could be inferred without a great deal of difficulty. Every conclusion in the library, for example, could (and perhaps should) be a feature; on every problem where they unify, the rule suggests a means of proceeding. The space of these features could quickly become very large; an improved capacity to learn mappings could eliminate some useless features, but there remains a lot of work to achieve this.

The more interesting problem, to us, is how compositional or even non-syntactic features could be learned. Pattern recognition and categorisation techniques might be useful here, but the most promising approach, it seems to us, would be to introduce some "hidden features" which are trained similarly to hidden layers in neural networks, with Hebbian learning and back propagation of credit assignment. This could go some way to learning an appropriate execution hierarchy. By continuing the neural network analogy, it might be possible to introduce a large number of such feature nodes, without any explicit meaning, which evolve together to develop ensemble codings potentially corresponding to more sophisticated features. The analogy between our `NetworkWeightList` executor and the structure of neural networks could perhaps be used to implement black-box neural network capabilities in an architecture which is ordinarily inspectable and compositional.

Learning Methods

The final component to problem solving control knowledge, in the feature-schema theory, is the schemas which apply. In Feasch with Isabelle, the schemas correspond very closely to specific rules, and it would be possible to construct standard forwards-rule and backwards-rule application schemas for each rule. Again, we would need a powerful mapping learning algorithm to filter out those which are not useful, but it at least suggests a way that methods could be learned.

One common technique for learning new methods is chunking (§2.1.5.i) old methods together. In theorem proving, this has been done by looking at common sequences of proof steps and compiling a Markov model of what steps are likely to follow a given set of predecessors (Duncan, 2006). By including features in the analysis, this model could focus its scope to patterns where methods have been chosen *for the same reasons*. We hypothesise that this information would remove noise from the analysis and lead to a richer set of generalisations.

Another approach to learning methods is top-down, by performing an inductive analysis on entire proofs (or proofs of subgoals). By including features in the abstraction of these "plans", we could again expect more reliable generalisations. By encoding these plans as modular schemas available for any purpose, more general schemas could perhaps be used to reason *about* the generalisation, applying it analogically, using it to infer new plans, or performing plan repair (*cf.* critics, §2.3.iii) — in the same framework as the proof step application. The availability of causal information (features) might furthermore be useful for constructing explanations for why such plans (or chunked tactics) are valid, increasing confidence in good plans and allowing spurious generalisations to be culled. (Muggleton, 1992)

Learning Methods without Learning Methods: Analogy

The top-down approach to learning methods could be very powerful, but we expect that it will also be very difficult. It might be possible to display the same "learned skill" without abstracting any methods. If past proofs and proof steps are stored as schemas in Feasch, the system could use "plan continuation features" to retrieve these steps without having any general plan or method schemas. Using analogical techniques, it might be possible to transfer proof steps from previous examples, adapting them and applying them to the current problem. No induction is necessary, and the user does not need to define any methods, but a relevant command could still be suggested interactively or applied automatically. This, it seems to us, would be an especially powerful capability, enabled by the use of features and schemas, which has not been possible with other control knowledge techniques.

9.2. Establishing a Computational Architecture

We have shown several ways that Feasch with Isabelle could be extended, through support for planning or learning. As these changes are not specific to Isabelle, they could be made in Feasch alone and thereby be available in Feasch as a general architecture for problem solving. What we have not shown, however, is that Feasch can be useful as such a general architecture. To show this, Feasch will have to be applied to numerous other domains — by multiple groups of users. Clearly this is well beyond the scope of the present project, but it seems to us that it could equally well have benefits in areas very different to theorem proving. There was nothing special we had to do to the Feasch system to use it for Isabelle, and as a framework for problem solving, we discovered it led to a different style of expressing control knowledge and to a unique set of benefits. We will be very interested to see whether it might have the same result in other problem solving research areas, and to this end we suggest two other candidate domains we would like to see explored: natural language and game playing.[34]

We will also be interested to see what implications for the Feasch approach are suggested by exploration in these other areas. Theorem proving led us to recognise the importance of planning, and the resulting practical insights have contributed to the shape of our general theoretical formulation, presented in Chapter 10. There are two particular areas where we feel our understanding — and our framework — is incomplete: how structural features are learned, and how explanation can feed back into the system to improve it. In the long-term, the measure of this approach will be the extent to which it aids understanding in many areas of problem solving.

9.2.1. Natural Language and the Importance of Structure

Learning useful structural features is one of the largest challenges in AI, and psychological theories of the processes by which people achieve this are scant. We anticipate that our system could help with this, by making structural features an integral part of its processing, and by having schemas as a single point of integration, so that "feature detectors" can be closely related both to "operators" and to "cases".

One prospect for a solution is providing a background process built-in to Feasch by which feature detectors are created automatically from encountered problems and solutions. These could be for surface features, so that family resemblances and associationist cueing can be facilitated for the important surface features, as well as structural features, on the basis of the structure that enables successful schemas to apply. One suspicion we have formed, in the course of this work, is that two orthogonal processes should govern this learning process: one is a periodic knowledge synthesis and "clean-up" to usefully restrict the focus of activity, as used in evolutionary programming (§2.2.2.iii) and possibly achieved by a utility ranking or by a

[34] While we hope to pursue some of these questions, we would be even more pleased if the reader has been moved by our review to investigate some of these ideas. Our system is available for such research, with code on-line and substantially more technical documentation (and comments) there than is included in this thesis.

dimensionality reduction; the other is an ongoing knowledge induction, where observing patterns in experience leads to the creation of new schemas, especially for detecting structural features.

Natural language understanding would be an interesting domain within which to explore this, because there is a rich body of training data available, CCG's, categorical associations (*e.g.*, parts of speech) learned by "clean-up", and relations between them (structural schemas, *viz.*, grammar) learned by induction over sentences where words are annotated with the hypothesised category features. To be sure, there has been a lot of research in this area, but we think there may be scope for Feasch to promote an investigation using features and schemas. The standard AI approaches described here, production rules and search in particular, have been spectacularly unsuccessful at learning grammar, and most computational approaches to nature language have used a variety of specialised techniques. If it is possible with Feasch, describing these approaches in a unified theoretical context would be certainly be useful, *e.g.*, for identifying synergies and possibly new ideas and means of learning structure. While we have doubts whether such an attempt would ultimately prove successful, we are sure that such a challenging would be a good exercise for exploring Feasch, highlighting its weaknesses and — possibly — revealing particular strengths.

9.2.2. Game Playing, Go, and the Importance of Explanation

Another active area of problem solving research where it would be attractive to apply Feasch is game playing. In early studies with the game Go, we found that that generalisations on the basis of actions alone was insufficient (Heneveld, 1999); this is a common finding, but there are few techniques for improving the quality of these generalisations. Including features in the analysis, we have noted, is potentially one such technique, and we would like to apply it to our original experiment in learning schemas for Go.

The most powerful machine technique for producing good abstractions seems to us to be explanation based generalisation (Barletta & Mark, 1988). Features, as we have discussed, can provide the efficient reasons why schemas are chosen; this is not necessarily the same as the final cause — why the schemas are good ones — but if the features regularly cue appropriate schemas, the reasons are likely to be the same or similar. Thus having the causal chain of "self-understanding" in place could lead to the prospect of being able to explain why the chains work, in a way not unlike the self-explanation effect in psychology (§1.1.3).

9.3. Psychological Experiments

One benefit which we hinted at, at the end of Chapter 1, is the potential use of this architecture for cognitive modelling. Specifically, it functions along the lines of several memory-based theories, using features for retrieval and schemas to perform the application, and so could offer a good framework in which to test these theories as models of human cognition. With regards to the very general notion of schemas, we recognise the argument that it can be self-defeating, but as discussed in §1.4, further high-level distinctions do not yet seem to be mature or supported by sufficient evidence. The use of a model would provide important specification in practice without restricting the purview of the theory.

Our system agrees with the theories by performing associationist retrieval, using a network of detectors that can identify both surface and structural features. Although these detectors are only executed serially in the applications in this thesis, this is due to current single-thread limitations of Isabelle, and the current code for Feasch supports concurrent execution of schemas. Running detectors in parallel represents a more psychologically plausible and computationally powerful basis for retrieval, on those machines and networks where massively parallel processing is feasible.

Our system has not yet been tested as a cognitive architecture, and although we have used some study of human expertise (*viz.*, features for solving integrals; §7.1.3) to build our computational approach, there is much more we would like to do to follow-up on our initial inspiration from cognitive theories. Experiments will be necessary to show that the present architecture is suitable for cognitive modelling, let alone that resulting models — using features to cue schemas — give good fits to human data. Conducting such experiments is a high priority for us, as we have noticed casually that Feasch does seem to solve problems as we do. With regards to calculus alone, there are at least three areas where this modelling could be investigated:

> **Task Performance.** Solving integrals. Novices and experts. Both aggregate and individual.
>
> **Learning.** How do subjects improve? Can it be modelled by a dualistic account of learning new features and better associations?
>
> **Tutoring Systems.** If Feasch can describe patterns of expertise, in an expressively inspectable way — as was suggested by many of the experiments presented here — then it may be possible to teach these patterns. If we find that our system can model a user's activity (on "Task Performance", particularly with respect to novices), an automated comparison against the system's model of expert behaviour could be used to identify the heuristic control knowledge the user is lacking. Because this knowledge is inspectable in Feasch, we hypothesise that it could be communicated to a user; if the user improves as a result, this would support its relevance to human problem solving and provide evidence of another benefit of it as an approach for computational problem solving.

We are exploring working with the LeActiveMath project (`http://www.leactivemath.org`), where a machine tutor combines natural language with a cognitive model to provide instruction in solving derivatives. We are hopeful that this context may provide the opportunity to carry out some of these experiments.

9.3.1. Modelling Experts at Solving Integrals

Solving integrals could serve as a good domain for cognitive modelling because it is sufficiently complicated to lead to variety among subjects but sufficiently narrow that it can be formalised — and to a large extent has been — in Chapter 7. We noted there that no machine system has been able to achieve near-expert performance along the lines of human mathematicians. By more closely analysing the process by which experts reason, this might be possible.

Feasch could be used to build a cognitive model of an individual's task performance by attempting to find the mappings which describe that individual's method selection on the basis of the features detected. The calculation for this could be similar to that used in Chapter 8, except here what is being learned is not the "right" solution, but one preferred way of solving the problem. There are, of course, many ways to solve a problem, and it would be interesting to explore the ability of a Feasch model to account for either an individual's choice or an aggregate choice.

This investigation could also afford an opportunity to develop our understanding of how features might be used at the plan level. In a solution attempt we have observed, the subject periodically changed focus along what seems to be a best-first strategy:

$\int (\sin x)(\cos x)\, dx$

1. by parts: $\sin^2 x - \int (\sin x)(\cos x)\, dx$
    4. by parts: $\int (\sin x)(\cos x)\, dx$
2. u-subst, $u_1 = (\sin x)(\cos x)$: $\int \frac{u}{\sqrt{1-(2u_1)^2}}\, du_1$
    3. u-subst, $u_2 = 2u_1$: $\frac{1}{4}\int \frac{u_2}{\sqrt{1-{u_2}^2}}\, du_2$
        5. trig-subst, $u_2 = \sin u_3$: $\frac{1}{4}\int \frac{\sin u_3}{\cos u_3} \cos u_3\, du_3 = \frac{1}{4}\int \sin u_3\, du_3$
        6. recognised: $= \frac{1}{4}\cos u_3 du_3$
7. trig-id, $sin2x$: $\int \frac{\sin 2x}{2}\, dx$
    8. recognised: $\frac{-\cos 2x}{4} + C$

Features such as "this problem state is worse than the one before" and "none of the methods I can think of seem promising" are examples of indications that a different state may be preferable. Discovering how this is done by people — and how this might be done by Feasch — would be a worthy accomplishment.

9.3.2. Challenging the Use of Production Rules

The biggest argument that will have to be made to advocate this approach in practice will be showing distinctive benefits over production rules. While we have argued for this on many grounds in the course of this exposition, an experiment which gives clear and direct evidence is highly desirable. One such experiment could be conducted with the Tower of Hanoi task, a staple of cognitive problem solving experiments.

We have observed that subjects performing the task will occasionally take a disk off one tower and then discover that it cannot be moved to any of the other towers. While this is hardly surprising, we note that the removal of that disk is disallowed in the basic production rules solution of Newell & Simon (1972): the operator production states that "if X is on top of A and smaller than any disk on B, move X from A to B", but the precondition is not valid. While the use of the precondition is computationally efficacious, it is psychologically wrong. This has been corrected in more sophisticated cogntive models (Anderson, 1993) in production architectures, by making the removal and placement two separate actions. Removing a disk has a precondition that the disk is at the top of the source tower, and placement has the precondition that the disk is not larger than a disk already on the target tower; in this model, the production that Newell & Simon classed as an operator is learned through composition of the two smaller rules.

One experiment we would like to pursue is to unfold this one more level, so that systematic errors are made with all of the supposed preconditions. In doing so, we hypothesise that the preconditions Anderson identifies could also be dissociated from the action. A precise formulation of a task to explore this is as follows:

> Instead of moving disks between three towers whilst obeying a size constraint, use coloured beads which are moved between three groups whilst obeying a corresponding spectral order constraint (*viz.*, the rainbow ordering). For instance, with seven beads coloured red, orange, yellow, green, blue, indigo, and violet, a yellow bead's removal is restricted to the case where it is in a source group with no red or orange beads, and its placement is restricted to a target group with no red or orange beads.

The preconditions that Anderson uses are ones which are easy for people to verify, possibly by "external cognition" (*i.e.*, the world does the work, trivialising the selection of the smallest disk by ensuring that it is always the one on top) or by frequently used perceptual processes (comparing size, such as when placing a disk). In our isomorph, these preconditions are much harder to verify. The logical precondition imposed by the problem definition is no longer a trivial task, and we suspect that people will be more likely to make mistakes.

Consequently, the preconditions that have been used in existing cognitive models — being on top of the source and not being larger than anything on the target — would have to be removed. We hypothesize that a production architecture model of cognition in this isomorph would use a single action, moving X from A to B, with no preconditions, and it would account for human performance (including improvement) solely through associative weights and chunking. Furthermore, although the spectral constraint will likely be more difficult for subjects to learn,

we suspect that their performance in learning the essential Tower of Hanoi solution pattern will be similar to performance by subjects and pre-existing models for the canonical task. If these points can be confirmed, we would show that the utility of `if ... then` constructs in these cognitive models is merely an artifice of domain simplification; in other words, we would show that *production rules model merely the logical structure in a problem domain* and not, as prior theories have claimed, the human cognitive process.

Chapter 10. Towards a New Paradigm

The breadth of the potential future work is daunting, but there are many reasons we are encouraged. It is daunting because it means there is so much left undone by the present research, but it is heartening because there is so much of interest left to do. That our approach has turned up so many interesting ideas we feel is some circumstantial validation of the initial idea. That idea, that features cue schemas for problem solving, is neither novel nor sophisticated; what we have done that is novel, however, is to use this idea to construct a conceptual framework for a problem solving system. In the domain where we have explored it, the achievements have been compelling, and we are thrilled by the fact that we can envisage so many extensions, further experiments, and other potential applications.

For us, the Feasch system as described so far represents the first step in exploring a larger abstract formalism for problem solving. We have been able to address many of the issues raised in the last chapter, in what we feel are elegant ways, at this abstract level. It is the product of substantial reflection and analysis, and its first offspring has proven useful in our experiments to date, but there is far too little evidence to argue that it is in any sense the "right way" to think about problem solving. It is purely speculative and philosophical — readers who strongly prefer scientifically grounded theories are recommended to skip directly to the conclusions, §10.2— but it has been very useful to us. There is a rich diversity of approaches to problem solving in computer science and cognitive science, but very little in the way of high-level synthesis. For us, this formalism conceptually unifies the diverse approaches, providing a paradigm in which they can be understood, contrasted, and ultimately combined. We hope it may be similarly useful for the reader, but we stress that there are no guarantees, only illustrations, and that this speculation is ancillary to the core hypotheses we have presented and tested.

10.1. The Feature-Schema Architecture

The classic general problem solving architecture has been that proposed by Newell and Simon (1972), presented earlier in §2.1.1and included again here:

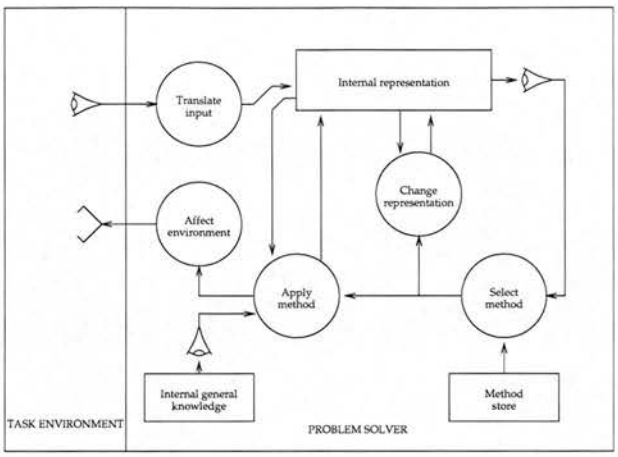

FIGURE 10.1: **General Problem Solver Organisation.** (repeated from figure 2.1 for reference; adapted from Newell & Simon, 1972)

At the time, this architecture represented state-of-the-art thinking, where methods are chosen and applied to a problem state within a state-space. It is still useful for problem domains where all methods are on the same level (*i.e.*, all methods directly changing the state space), but in large problem solving systems, this is rarely the case. Planning, for example, is a challenge to express in this diagram, for planning strategies do not act on the problem state but change focus within the problem's state-space. Newell and Simon discuss this, presenting a two-level formulation and recognising the need for this interplay between them. They note that there may be "new solution possibilities or demands that cause the problem solver to interrupt its current activities to try different ones", but their architecture fails to account for this interplay.

Their architecture also makes an explicit divide between "general knowledge" and "method store". This is questionable because methods are themselves a form of knowledge, the physiological evidence for this split is ambiguous (§1.2.3), and the same processes are used to retrieve both. Furthermore, general knowledge will almost certainly influence the method selection, although no provision for this is made in the diagram. Even the distinction between "method selection" and "method application" is misleading, for in all cases the method selection will itself be performed by some method, and in most models, there will be many methods which, when applied, have the effect of selecting other methods.

Method-chaining and general knowledge play a crucial role in "interpretation", which they define as the process of fitting a situation to a method. Apart from the distinct initial "translation" step in their model (which will use many of the same methods and techniques), they imply that these mechanisms are integral in "the structure of the method". In practise, they are needed for the "selection" process as well, and more often than not they are implemented as distinct methods (such as unification) which will be used by other methods.

Everything is a method, in the sense that for most problems solving systems, the most *interesting* aspect is the method (or procedure) by which it works. Newell and Simon write:

> Thus the theory focuses on the method: a collection of information processes that combine a series of means to attain an end, or at least to attempt to attain an end.

The distinctions between "translation", "selection" and "application", and between "general knowledge" and "method store" are often not relevant, and the distinctions drawn in figure 10.1 seem to us to provide more obfuscation than clarity about how actual problem solving implementations are operating.

10.1.1. The Schema Model

With this in mind, we recast the diagram of Newell and Simon to conflate everything from the diagram (apart from the problem state, input, and output) into a single entity labelled "apply schema", shown in figure 10.2. Schemas can be production rules or cases, or methods, strategies, scripts, frames, or heuristics. If there are other problem solving techniques that we do not yet know about, they can be included as schemas as well. It is deliberately completely open-ended, but at the highest level of abstraction it seems better to ignore the numerous distinctions which have been drawn (and which often, in practice, obscure a system's behaviour). We will refine this "apply schema" entity later, after describing the rest of our high-level model.

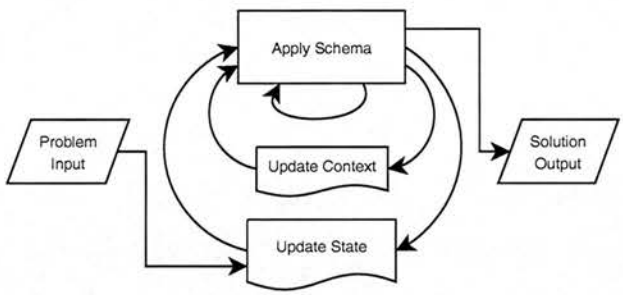

FIGURE 10.2: **The Schema Model**. Newell and Simon's General Problem Solver model can be recast as above to make use of context and to generalise over methods, cases, and strategies.

The one new entity introduced in the diagram is that of "context". This is related to the idea of "change representation" from figure 10.1, but substantially broadened. Although the notion of "change representation" is attractive from a Gestalt psychology point of view, in a logical sense it is unsatisfactory because *any* change to a problem state involves changing something in its representation. It is often a subjective judgment whether the *representational system* is changed or the *problem state* is changed (within the same representational system); and in practice it makes very little difference which is changing. What is vital, however, is being able to record information associated with a state but not integral to the state itself and to pass this information between schemas. "Change representation" hints at this importance, but something more general is needed to capture the complexity of altering and enriching the description of a problem.

In our framework, the "context" associated with a state can include a description of what schemas have been tried, what schema has just been applied, or what schema should be applied next. It can also include derived information about the problem state, such as a diagram (Winterstein, 2004), a mental model (Johnson-Laird, 1983), or meta-level reasoning information (Bundy, 1983). It may also include contextual information that guides the interpretation of a problem, such as whether it is a a formal logic problem (as was originally intended in the Wason Selection Task, §1.2.1), a situated problem (Gick & Holyoak, 1980), or a deontic problem (Cosmides, 1989).

To clarify the distinction between "Update State" and "Update Context", we define that the state is a concise representation of the minimal or near-minimal information needed to uniquely capture a situation in a problem. Changes to the state change the problem situation in a significant manner, and they can only be effected by operators specified by the problem domain. Contextual information, on the other hand, has no effect on the problem state; it can be updated by the schemas without restriction by the domain, recording any information which a schema finds (often derived from the problem state) which might be useful to it or other schemas. On an implementational level, the separation of context encourages control strategies and heuristics (implemented here as schemas) to make the information that they derive and use explicit, so that it can be re-used if the state is revisited or if another schema could benefit from the same information, and so that it can be inspected by an observer and more easily discussed. This separation also allows methods to gather a large amount of information without changing the problem state, simplifying both the number and the size of states encountered in the course of solving a problem.

The benefit of this formalism is that it more generally describes what actually happens in the vast number of problem solving systems. It begs the question of what representation is used, but we feel this is another benefit[35]: this should be a design choice the user can make, depending on what is appropriate for the domain at hand or the goal of the model. The user should not be forced by the architecture into using a representation that is not natural, and as yet there is no standard, universally applicable representation. Our implementations in this thesis have used predicate logic, but this aspect is purely "below the line" of what is theoretically important.

Various AI systems and cognitive models will often use their own specific representations and vocabularies; they may use different representations for the problem state, the context, and the schemas. These systems vary in the choice of representation — and they have enormous variation in the "apply schema" component, discussed next — but the three-part split, of state, context, and schemas, applies to nearly all of them. It is for this reason that this high-level model is interesting, despite the ambiguity of the component called "schema".

Types of Schemas

Most systems and models draw sharp distinctions within the category we call schemas, using entities referred to by various names: methods, heuristics, rules, cases, productions, experience, control knowledge, tactics, plans, etc. In our diagram, so far, we have conflated everything that does work into "schemas". We have — intentionally — left this category very general, in order to highlight that all such entities (methods, strategies, cases, or productions) used in problem solving approaches fit a common pattern: either they change the problem state, they change the context, or they are used by other schemas. As soon as this category is restricted, *e.g.*,

[35] Leaving the user completely unaided with respect to representation is not in their interests, and we provide support for a common standard — predicate logic — as used throughout this thesis. This representation has its limitations, however; our claim here is that it is a benefit of our framework that it is not dependent on this choice of representation.

by insisting that productions be used, the class of problem solving techniques which the model describes is reduced. Keeping it vague — at the highest level — establishes a framework for a modular, object-oriented architecture design, which although still general, can become useful once specific types and instances of schemas are identified. There are many different opinions about what types of schemas should be distinguished, but by using this high-level model, these different types can be accommodated in the same system.

We begin with three types of schemas which we found were commonly used in AI systems and cognitive models.

> **Operator Schemas.** The lowest level of schemas are those which directly change the problem state. Often these are tightly constrained by the problem domain, and as a result are very domain-specific, such as "move block X from Y to Z" in the STRIPS planner (Fikes & Nilsson, 1971). Some domains are very common, however, giving rise to some very familiar Operator Schemas. Logic is one example, where we would would expect Operator Schemas for *e.g.*, the formal rule of *modus ponens* (forward-chaining), "if we know $P \Longrightarrow Q$, and P is in the problem state, then add Q to the problem state".
>
> **Experience Schemas.** Another major class of schemas consists of those which simply record knowledge about a domain. They do not say anything about how the knowledge is applied, but may be used by other schemas to transform the problem state or enrich the context. A statement of the form $P \Longrightarrow Q$, such as "If someone is a man, then he is mortal", is one example of this type of schema, and might be used by a Forward Chaining Operator Schema. Many production rule systems encode knowledge in this type of schema, such as declarative knowledge in expert systems, and axioms and lemmas in theorem proving. Case-based reasoning is another domain that uses what we describe as Experience Schemas, either entire records of a specific experience or partially abstracted records. In cognitive psychology, this type of schema includes those identified by Chi *et al.*'s for physics models (1981) and in various folk worlds (see §1.1).
>
> **Guidance Schemas.** The largest and most interesting category of schemas are those which select Operator Schemas and guide how they will apply. Often, these will use experience schemas as a guide, as in *matching schemas*, such as predicate logic unification or structure-mapping analogy (Gentner, 1983). Other times, guidance schemas will be a defined order in which other schemas will be tested and applied, such as tactical expressions in theorem proving ("use Method1 then Method2 or Method3, or else Method3 then Method4 repeatedly"; as in §2.1.4), or the Prolog approach of sequentially looking at definitions in a Prolog program. More generally, strategies such as those described by Polya (§1.3) are contained in this class, including, in theorem-proving, the technique of contradiction, and in any goal-directed domain, working backwards and subgoaling. Heuristics are Guidance Schemas which may rank other schemas in the domain, storing the scores in the context and then choosing the best one, as in cost-benefit schemas such as means-end analysis and hillclimbing, and in "availability" schemas where rankings of other schemas will be based on the recency, frequency, and success rate of past applications (Tversky & Kahneman, 1973).

Many examples of problem solving techniques can be described in terms of these schema types and some representation of both state and context. We emphasise, however, that these categories are approximate, and schemas can often be implemented in one or another category depending either on a particular system's requirements or a particular developer's preference for a given domain. What is useful about the Schema Model is that it introduces vocabulary to distinguish between various implementations. The essential distinction we draw between these three types of schemas is that *operator schemas* act on the problem state, *experience schemas* store data but no code for how they are used, and *guidance schemas* can run, often updating context or using other

schemas. The distinction between these types of schemas often corresponds to important design choices about how problem solving information is represented. From the intent of modular design, we recommend that schemas be put into the briefest, most general, and most reusable form. This can sometimes be a difficult choice to make; we suggest, following principles of modular software design, that new schemas should use other schemas as much as possible rather than duplicate functionality that exists elsewhere.

One benefit of recording knowledge in the most modular, reusable way is that these schemas may then be analysed and abstracted to form new schemas. This introduces a fourth type of schema:

> **Learning Schemas.** Learning may happen by making new experience schemas from problem solving attempts, by creating generalised experience schemas from several specific experience schemas, by creating a guidance schema from experience schemas, by forming macro-operators from other operator schemas, *inter alia*.

There are very many learning techniques described in the literature, but again an advantage of incorporating them as schemas in this model is that their activity can be described in the same formalism as problem-specific solving activities. In a system, it could run concurrently and, if implemented in a modular way, techniques could potentially be applied, easily and off-the-shelf, to knowledge in other problem domains in the same system.

10.1.2. Planning and Search

Searching and planning are extremely important aspects of problem solving which are not shown in the Newell and Simon formalism, though they say vaguely that they can be treated as "sub-problems". In the Schema Model, they can be quite neatly expressed by considering that planning takes place by applying Planning Schemas on a space of all possible problem states (the state-space level), which pass problem states to schemas which act on problem states (the problem-state level). This can be viewed as a nested version of figure 10.2, shown in figure 10.3.

For depth-first search, the introduction of a "space of all states" is not needed; it can be accomplished by schemas storing a copy of the current state before calling other schemas, iteratively, on updated states. It is cleaner, however, for the schemas to run quickly without using recursion (storing information about what has been done, and how to continue, in the context of the state), and this requires the introduction of a state-space (sometimes called the problem space, search space, or planning space). This is cleaner because control is returned quickly to the outermost schema, which will typically have the greatest flexibility, in selecting schemas to apply. This follows another principle of modular design, that the control architecture should be shallow, analogous to an "event-handling loop", and should allow for interruption as recognised by Newell and Simon (§10.1). This can correspond to psychological accounts of executive schemas such as the danger schema: if engaged in the Tower of Hanoi task when a fire breaks out, a subject will show behaviour altogether different than any of the models in §9.3.2 predict.

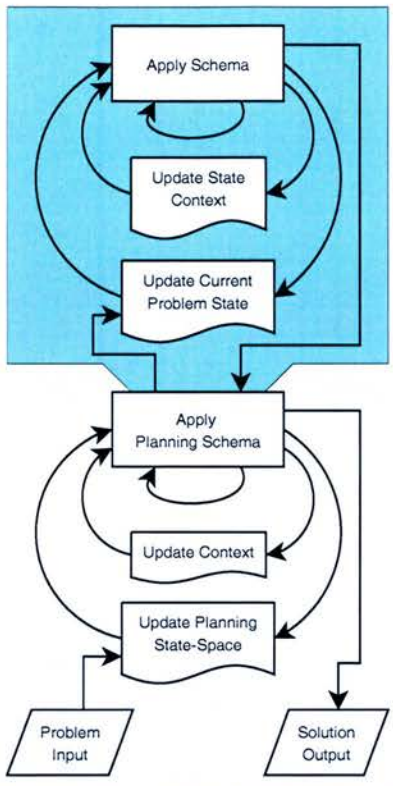

FIGURE 10.3: **Planning in the Schema Model**. Planning (and its frequent isomorph search) can be expressed as a two-tiered Schema Model where Planning Schemas act in a state-space (not shaded) and can call other schemas to act in a particular problem state (shaded).

What the introduction of a state-space does is to allow schemas to analyse the entire space of problem states. This is necessary for best-first search to identify which state seems best at any given point, and for more advanced planning schemas such as A^* search (Pearl, 2000) and hierarchical task network planning (Nau *et al.*, 1999). It is also essential for such activities as loop avoidance and identifying duplicate states, as well as more sophisticated techniques like identifying analogous states or state sequences, recognising portions of state-space which are steadily getting worse ("ballooning"), and pruning entire regions of the state-space at one time (as in the N-queens problem). This leads to the introduction of another category of schemas:

> **Planning Schemas.** Specialised schemas for working in a "state-space", the tree or graph of problem states encountered during a problem solving activity, usually passing problem states from the "state-space level" where the Planning Schemas apply to a distinct "problem-state level". Examples of these schemas include search strategies / planning techniques (depth-first, iterative deepening, best-first, A^*, minimax / α-β, HTN) and optimisation techniques (identifying duplicates, pruning unproductive regions of state space).

As shown in figure 10.3, Planning Schemas will often select a problem state and pass it to a non-planning schema in the problem-state level (shaded in the figure); this will often result in a new state (which the Planning Schema will add to the problem space) or in updated context for that state. Planning Schemas often will also note the most recent state found in the context at the state-space level. This information is useful because some Planning Schemas do not simply select a state and pass it to the problem-state cycle, but instead act directly on the state-space,

and sometimes the context of what has been done recently is important in these optimisation techniques. A loop avoidance schema, for example, can look at the recently introduced state and determine whether it is a duplicate of any of its ancestors in the state-space, and mark it as failed if it is. Another example of Planning Schemas which act on the state-space are critics (Ireland *et al.*, 1999), which will look at a recent failed schema application in the domain of theorem proving and determine whether some repair can be effected.

In §9.1.4, we suggested that planning could be implemented using features which persist in the context from one state to the next, and where these features cue the continuation of the plan. The analysis here suggests that an alternative implementation might have distinct state and context representation for the plan level and the problem level. This corresponds better to how planners are implemented, including IsaPlanner, and we prefer it (although we find a further improvement below).

This modular approach to planning allows different planners to be easily swapped by a developer; if one planning technique is very efficient in most cases but, say, in certain situations gets trapped in a local extrema, a developer might try to use that efficient planning schema to solve a problem, and then, if it does not succeed, manually apply a more powerful but difficult one. Or she could develop a "mixed" Planning Schema, which automatically tries the efficient schema first and the more powerful one only if necessary. Looking back to the class of Learning Schemas, by allowing them to be provided as modular entities in a system, our formalism could permit them to be applied to planning schemas so that such a mixed schema could itself be learned automatically.

Situated Planning

There is an alternate way to formulate planning in the Schema Model: the possible states encountered in search (the shaded region in figure 10.3) can be viewed as part of the context of the state-space. This becomes useful when the entire problem solving system is embodied, *e.g.*, in a robot, and where it will ultimately makes changes in some external environment. In other words, the "Update State" component of figure 10.2 could mean to perform an action in the real world; in such a system, the planning must be done in a virtual environment, and the actions effected after a plan is created with sufficient confidence.

In this model, both the "state-space" and the internal representation of the current "problem state" are aspects of the context, and the different types of schemas (Operator, Guidance, and Planning) act on relevant entities either in the state or in the context. Situated problem solving will typically use virtual forms of operator schemas to manipulate an internal representation of the problem state, and the corresponding physical operator schemas will be used to modify the outside world after a plan has been found. Grounding problem solving systems in real-world situations can pose additional challenges (such as reconciling the virtual representation with what actually happens), but the results tend to be more robust and more faithful to theories of cognition

than systems which act purely in artificial problem domains. The real-world applicability of such systems is an additional, practical reason to address these concerns.

The entire virtual representation, in such systems, can be considered as "context" which the system uses to enrich its understanding of the actual world. However, within this internal context, not making the problem state and the state-space explicit can obscure the importance of a particular focus at a particular level of problem solving. For this reason, we distinguish between "problem state", "state-space", and the "outside world". Depending on the focus of the model as determined by schemas, the "Update State" component can refer to any of these levels. This could be illustrated in a three-tier "tower" of schema models with the outside world set beneath planning and the problem state shown in figure 10.3; another representation, paying heed to the observation that "state" at one level is merely part of context at the lower level, is to use three levels of cycles, hanging off the "Apply Schema" component and illustrating also the relationship between the contexts of the different levels. This is shown in figure 10.4.

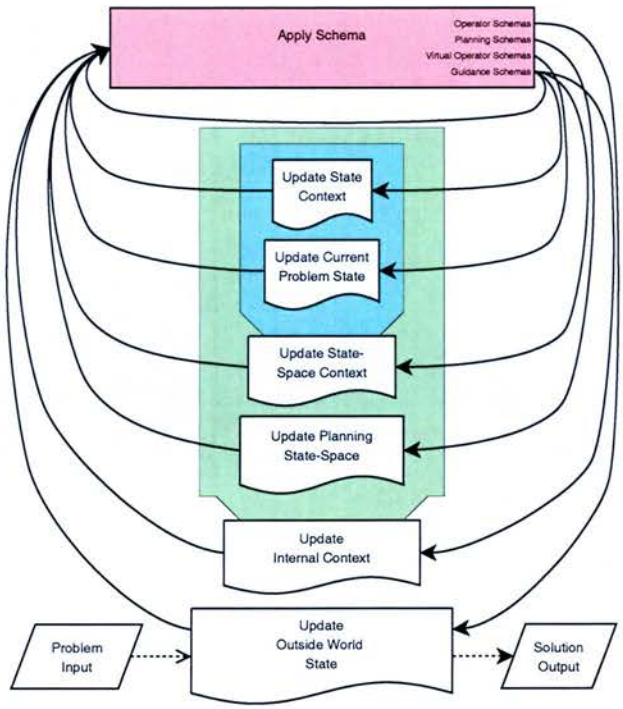

FIGURE 10.4: **Situated Planning as a Multi-Cycle Schema Model.** Planning situated in a real-world application can be viewed as a multitude of cycles where internal states are part of the context of the surrounding state.

Although the "tower" visualisation shown earlier is simpler, we prefer the "multi-cycle" formulation shown here because we feel it more naturally represents situated systems and cross-level events such as interruption/refocussing (such as if a fire breaks out). Both, however, convey the need to support problem solving at multiple levels, both make these levels explicit (for the examples of planning and situated problem solving), and both illustrate how these can be achieved within our formalism.

10.1.3. Features in the Schema Model

Our formalism has so far not introduced the key ingredient in the problem solving system we have developed in the course of this thesis: features, *i.e.*, aspects of a problem situation which may not correspond to the raw units of a problem representation but which play a role in determining the response. This is because we wish to show that the formalism is useful not just for describing the Feasch system, but as a widely-applicable means to describe many problem solving systems.

We relate the Feasch system to our formalism by introducing a particular Guidance Schema, *viz.* the "Feasch Executor Schema", which follows the three-step process of detecting features in the problem state, retrieving schemas based on the features, and successively applying the most strongly cued schemas. This schema runs as the top-level "apply schema" box in Figure 10.2, as shown in Figure 10.5.

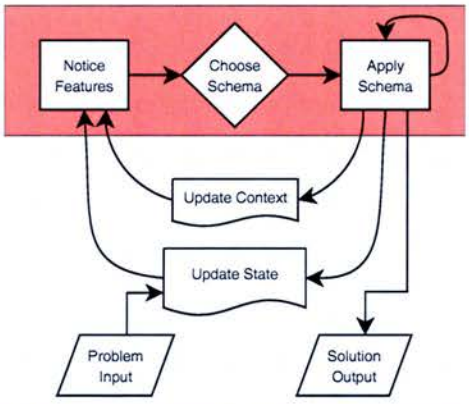

FIGURE 10.5: **The Feasch Executor Schema.** A general schema guides the selection of schemas based on features and a feature-schema map.

This correspond to the `IExecutors` introduced in Chapter 3. One more category of schemas follows:

> **Feature Detector Schemas.** New features can be detected, by analysing the current state and the associated context, and added to the current context. The Executor will use these features to cue other schemas.

Planning and Situatedness with the Feasch Executor Schema

If we combine the Feasch Executor Schema in Figure 10.5 with the multi-level formalism in Figure 10.4, an interesting solution emerges for the "statefulness and search" limitation of the Feasch with Isabelle system (§9.1.4.i). Feasch might act at the problem state level (as Feasch with Isabelle does currently), or at the state-space level (for planning and search), or even at the outside world level (when the system is capable of interacting with the external world). Features can be used to keep track of the internal notion of which level is being used, ensuring that schemas appropriate there are selected. For instance, a feature in the state-space context might indicate that "I just made state N"; this might trigger a Guidance Planning Schema for depth-first, which could suggest the feature "look at problem state N", which would in turn focus the current problem

state on what the planning state-space has recorded as state N. The Feasch Executor would then generate and respond primarily to that problem state and features in its context; say that after further examination "state N does not look good", the context of the state-space could be updated with a feature to that effect. This, together with the "I just made state N" feature, could trigger back-tracking. If, backtracking a level, the parent state M also looks bad, a feature might be added to the context saying that "the whole branch from state M looks pretty bad", and the model might then abandon depth-first search and try a best-first search or a plan script, possibly one cued by features detected in states M and N.

In general, a planning schema will update the context with an indication that a particular hypothetical problem state has been identified. The Feasch Executor will respond to that feature by focussing feature detection and schema selection on that state; if the activation level of these schemas is low, however, features from the planning level might result in the activation of a Planning Schema. If features stop giving strong support for one Planning Schema (*e.g.*, depth-first search), an alternative schema (*e.g.*, best-first search) might be activated; this allows planning techniques to be mixed, with different ones applied as different situations dictate. Of course, if other features are noticed — such as a fire breaking out — it may send a very strong cue to schemas at an even higher level — such as an "evacuate the building" schema. State duplication is a more likely example, and in any good search algorithm in Feasch, such a Feature Detector Schema would be included at the planning level, cueing appropriate remedies such as to abandon that branch of the search tree or update the internal representation so that multiple paths to the state are recorded in a state-space graph. In a more elaborate example, a feature at the problem state level might recognise a feature that "we're close but it doesn't quite work" (*e.g.*, "we're in the right order but on the wrong target pole" in Tower of Hanoi); this might trigger a known Planning Schema for plan repair, or if the history of steps is also described as features at the planning level, a specific past experience might be triggered (*e.g.*, where, in the past, the same or a similar pattern led to the tower being built on the wrong pole, and the problem was resolved by changing the pole used on the initial move).

10.1.4. Relation to Other Work

What we find useful about this formalism is that it allows many techniques to be represented using the same vocabulary and the same theoretical architecture. Different techniques for planning and search can be represented as schemas which can invoke any Guidance Schema or Operator Schema. A plan might call to apply a production rule, and if it fails it might invoke the Feasch Executor. Conversely, the Feasch Executor might determine that a particular plan is called for, or a certain low-level action, or even a case-based reasoning schema, if available. Strategies for problem solving can be conceptualised in a unified framework, and although this is far from being an implemented system where these are all compatible, it is a necessary first step. There may be other ways of synthesising these various techniques, but we have shown at least one way

that such an attempt could be made. Showing the compatibility of, *e.g.*, analogy within ACT-R, was an inordinately difficult — and ultimately unconvincing — undertaking (§1.2.3). While this armchair analysis offers even less in the currency of evidence, it offers substantially more in the currency of modularity and flexibility. It provides a framework for problem solving in which components for search, planning, learning, and Feasch can all coexist and interact.

The initial steps in this paradigm have revealed a ripe area of research questions, and given that there is always more to be understood in science, one of the best measures of a theory is how much new research it generates. There has been a wealth of new techniques in cognitive science and Artificial Intelligence since Newell & Simon described their General Problem Solver (1972) — many of them precisely because of their introducing a general architecture and specific challenges; but there has been a dearth of dramatically different rival architectures, grounded in philosophy and psychology and applied to psychology and computation. SOAR and ACT-R are architectures so grounded, but they continue what is essentially the same production rules approach. With Feasch — and with the further abstraction herein — we shape a true alternative, This paradigmatic view of problem solving offered insight into how planning and search could be incorporated into the original system we designed, where features cue schemas in a stateless problem solver. We found the resulting ideas much more attractive, in our eyes, than the solution we originally imagined (*i.e.*, the multi-level formalism seems much more principled than the use of persistent features, §9.1.4.i), and we look forward to extending Feasch along these lines. How well this extension pans out remains to be seen, but it is merely one suggested research direction we have found (admittedly our favourite). What is more important, as the exploration of Feasch with Isabelle showed, is that the cross-fertilisation of problem solving ideas can lead to interesting systems with novel, useful properties. We hope that this formalism can encourage and support such creativity.

10.2. Concluding Thoughts

There is indisputably much more work required to show either that the framework we present is useful as a general computational approach or that it has clear relevance to modelling psychological problem solving. We are hopeful that many of these ideas are both interesting and feasible, and we believe that the initial study of the ideas in specific problem areas has been promising. What we have tried to demonstrate in this initial investigation is that making features explicit — and emphasising structural features in particular — yields an interesting approach to problem solving, and specifically that:

> **It is possible.** By creating the Feasch system and showing that a noughts-and-crosses player can be easily implemented therein (Chapter 3), we established our possibility hypothesis, *(PH) features can be the foundation of an AI problem solving system.*
>
> **It is uniquely different to other approaches.** As used in Isabelle, we discovered many expressivity properties of using features and schemas which are not true of other ways of encoding control knowledge. These are that features *(EH1) provide reasoned*

explanations for method selection, e.g., for use in generating natural language proofs with causal information; (EH2) enable useful guidance for interactive theorem proving; and *(EH4) contribute to modular design of control knowledge.* The one hypothesis which was found to be difficult with Feasch — *(EH3) encoding control knowledge for automated theorem proving* — is the only one which strongly holds for tactics and IsaPlanner.

We have contrasted our approach with control knowledge techniques which are representative of widely-used approaches (in our view, they are among the best in the literature, because they permit the expression arbitrary tactics, and can be developed alongside the mathematical theories). Crucially, the underlying Feasch system did not need to be changed in any way to permit our implementation or to demonstrate the expressivity properties. As a result it seems likely that these expressivity properties could generalise to other domains.

Of specific applied benefit to the Isabelle community, Feasch with Isabelle yields a means of expressing heuristic control knowledge which is easy to write and easy to understand. If a developer formulates this control knowledge, in addition to new definitions and theorems, our system can use it to assist the user in at least four ways: semantically, by showing features detected in a problem space and the commands these features typically cue (§5.2); interactively, where a user can "point-and-click" on suggested methods (§5.3.1); automatically, through the "fast-forward" button (§6.2.2); and semi-automatically, with the "auto-run threshold" (as well as any modifications a user might choose to make) determining which high-confidence commands are applied automatically and which more questionable commands are left for the user to apply at her discretion (§7.2.1). We showed how this is difficult with many standard techniques; we note further that these heuristics support parallel execution, which is also difficult with most standard techniques but increasingly important in real-world computations. This led to the first major benefit we claim for our approach:

(BH1) The inspectability and modularity properties of feature-based control knowledge are beneficial to multi-agent theorem proving.

We then showed how, by defining only the features and methods relevant to differentiation, our system was able to discover the mappings which comprise the decision procedure for the domain. This showed the second and final major benefit we claim:

(BH2) Feasch allows the use of learning mechanisms not applicable to other systems, simplifying the encoding of control knowledge and yielding improved performance.

In the course of this thesis, we have developed a problem solving framework Feasch; extended the public-domain theorem proving interface Eclipse ProofGeneral; created a powerful new means of expressing control knowledge for the Isabelle theorem prover, with a GUI and its own Feature Description Language, using Java and ML; completed several proofs for an Isabelle theory of calculus, providing control knowledge in our system for differentiation and integration; and devised a learning agent for this control knowledge. Each of these components is available from the project web page, `http://feasch.heneveld.org`, and we expect several will be useful

to a wider audience. Projects — some by ourselves and some by third-parties — are under way using one or more of the components we have developed.

We have also identified several further areas and potential benefits: extending the capabilities of Feasch with Isabelle and the support for it; assessing Feasch in other problem domains; and, if such examinations yield further encouraging results, trying to extend the power of Feasch as a general framework. Apart from the experimental evidence of the usefulness of the approach to theorem proving in Isabelle, we believe that our theoretical considerations may also make a substantive contribution to the theory of problem solving. We have generalised a cognitive theory, where features cue the retrieval of schemas, shown that it applies very usefully in one concrete and difficult domain, and argued that it encompasses, in theory, a great many problem solving techniques. Showing applicability in many domains, far beyond theorem proving, will be essential to establish that the proposed architecture is good generally, and consequently we will be particularly interested to see what happens when attempts are made to extend its capabilities for learning (§9.1.5), planning (§9.1.4, §10.1.3.i), and cognitive modelling (§9.3). Having found certain benefits in the domains where it has been tried, *viz.*, for modular control knowledge development and for learning, however, we have convinced ourselves — and we hope also the reader — that it provides a promising perspective to the problem of problem solving.

Bibliography

Alexoudi, M., C. Zinn, & A. Bundy. (2004) English summaries of mathematical proofs. In *Workshop Programme at the 2nd International Joint Conference on Automated Reasoning*. `http://homepages.inf.ed.ac.uk/zinn/Publications/AlexoudiZinnBundy.pdf`

Allen, J. (1995) *Natural Language Understanding*, 2nd ed. Menlo Park, CA: Benjamin/Cummings.

Anderson, J. R. (1985) *Cognitive Psychology and its Implications*. New York: W. H. Freeman.

Anderson, J. R. (1993) *Rules of the Mind*. Hillsdale, NJ: Erlbaum.

Anderson, J. R., & C. Lebiere, eds. (1998) *The Atomic Components of Thought*. Hillsdale, NJ: Erlbaum.

Anderson, J. R., & C. D. Schunn. (2000) In R. Glaser, ed., *Advances in Instructional Psychology*, vol. 5. Mahwah, NJ: Erlbaum.

Anderson, R. C., R. J. Spiro, & W. E. Montague, eds. (1977) *Schooling and the Acquisition of Knowledge*. Hillsdale, NJ: Erlbaum.

Apt, K. (2003) *Principles of Constraint Programming*. Cambridge: Cambridge University Press.

Arnheim, R. (1954/1974) *Art and Visual Perception*. Berkeley: University of California Press.

Arthur, D. (2000) Calculus teaching notes. Technical report, Mathematical Teaching Organisation, University of Edinburgh.

Aspinall, D., *et al.* (2006) ProofGeneral. `http://proofgeneral.inf.ed.ac.uk`

Barletta, R., & W. Mark. (1988) Explanation-based indexing of cases. In *Proceedings of the Seventh National Conference on Artificial Intelligence*.

Barsalou, L. W., J. Huttenlocher, & K. Lamberts. (1998) Basing categorisation on individuals and events, *Cognitive Psychology 36*:203-272.

Bartlett, F. C. (1932) *Remembering*. Cambridge: Cambridge University Press.

Ben-Zeev, B. (1988) The Schema Paradigm in Perception, *Journal of Mind and Behavior 9(4)*:487-514.

Berlekamp, E., & D. Wolfe. (1994) *Mathematical Go Endgames*. San Jose: Ishi.

Bloom, B. S., & L. J. Broder. (1950) *Problem-solving processes of college students*. Chicago: University of Chicago Press

de Bono, E. (1985) *Six Thinking Hats*. Boston: Back Bay Books.

Boole, G. (1854) *An Investigation into the Laws of Thought*. London: Walton and Maberly.

Borges, J.-L. (1964) *Labyrinths*. London: Penguin.

Bozulich, R., ed. (1992) *The Go Player's Almanac*. San Jose: Ishi.

Brachman, R., & H. Levesque. (2004) *Knowledge Representation and Reasoning*. San Diego, CA: Morgan Kaufman.

Braine, M. D. S. (1978) On the relation between the natural logic of reasoning and the standard logic, *Psychological Review 85*:1-21.

Brown, T. (1828) *Lectures on the Philosophy of the Human Mind*. Edinburgh: Tait.

Bruner, J. S., J. J. Goodnow, & G. A. Austin. (1956) *A Study of Thinking*. New York: Wiley.

Buchberger, B., & A. Craciun. Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema. In F. Kamareddine, ed., *Proc. of the Mathematical Knowledge Management Workshop*, Edinburgh, 25 Nov 2003. Electronic Notes on Theoretical Computer Science, volume dedicated to the MKM-03 Symposium. Elsevier.

Buchberger, B., and the Theorema Group. (2006) Theorema. `http://www.theorema.org`

Bundy, A. (1983) *Computer Modelling of Mathematical Reasoning*. New York: Academic Press.

Bundy, A. (2006) Lecture delivered at Cork. Personal communication.

Bundy, A., L. Byrd, G. Luger, C. Mellish, R. Milne, & M. Palmer. (1979) Solving mechanics problems using meta-level inference. In *Proceedings of IJCAI-1979*, 1017-1027.

Bundy, A., A. Stevens, F. van Harmelen, A. Ireland, & A. Smaill. (1993) Rippling: a heuristic for guiding inductive proofs, *Artificial Intelligence 62*:185-253.

Burmeister, J. (1999) Research Page. `http://www.psy.uq.edu.au/~jay/`

Buswell, G. T. (1956) *Patterns of Thinking in Solving Problems*. Berkeley: University of California Press.

Chalmers, D. J., R. M. French, & D. R. Hofstadter. (1992) High-level perception, representation, and analogy: A critique of artificial intelligence methodology, *Journal of Experimental & Theoretical Artificial Intelligence* 4:185-211.

Chase, W. G., & H. A. Simon. (1973) Perception in Chess, *Cognitive Psychology 4*.

Cheng, P. W., & K. J. Holyoak. (1985) Pragmatic reasoning schemas, *Cognitive Psychology 17*:391-416.

Chi, M. T. H. (1993) Current Contents. `http://www.garfield.library.upenn.edu/classics1993/A1993LZ47400001.pdf`

Chi, M. T. H., M. Bassok, M. W. Lewis, P. Reimann, & R. Glaser. (1989) Self-explanations: How students study and use examples in learning to solve problems, *Cognitive Science 13*:145-182.

Chi, M. T. H., P. J. Feltovich, & R. Glaser. (1981) Categorization and representation of physics problems by experts and novices, *Cognitive Science 5*:121-152.

Churchland, P. S., & T. J. Sejnowski. (1992) *The Computational Brain*. Cambridge, MA: MIT Press.

Clocksin, W. F., & C.S. Mellish. (1994) *Programming in Prolog*, 4th ed. New York: Springer-Verlag.

Collins, A., & M. R. Quillian. (1969) Retrieval time from semantic memory. *Journal of Verbal Learning & Verbal Behavior 8*:240-247.

Cooper, R. & J. Fox. (1998) COGENT: a visual design environment for cognitive modeling, *Behavior Research Methods, Instruments & Computers 30*:553-564.

Cooper, R., P. Yule, J. Fox, & D. Glasspool. (2006) The COGENT Project. `http://cogent.psyc.bbk.ac.uk/`

Cosmides, L. (1989) The logic of social exchange: Has natural selection shaped how humans reason? Studies with the Wason selection task, *Cognition 31*:187-276.

Csikszentmihalyi, M. (1990) *Flow: The Psychology of Optimal Experience.* New York: Harper and Row.

Dixon, L. (2006) IsaPlanner. `http://isaplanner.sourceforge.net/`

Duncan, H. (2005) *The use of data mining for the automatic formation of tactics*, PhD thesis, University of Edinburgh.

Duncker, K. (1926) A qualitative (experimental and theoretical) study of productive thinking (solving of comprehensible problems), *Journal of Genetic Psychology 33*:642-708.

Duncker, K. (1945) On problem solving, *Psychological Monographs 58*:(5).

Epstein, S. L. (1994) For the right reasons: the FORR architecture for learning in a skill domain, *Cognitive Science 18*(3): 479-511.

Ernst, G. W., & A. Newell. (1969) *GPS: a case study in generality and problem solving.* New York: Academic Press.

Estes, W. K. (1991) Cognitive architectures from the standpoint of an experimental psychologist, *Annual Review of Psychology 42*:1-28.

Evans, J. St. B. T., S. E. Newstead, & R. M. J. Byrne (1993) *Human Reasoning: The Psychology of Deduction.* Hove, UK: Erlbaum.

Ewert, P. H., & J. F. Lambert. (1932) The effect of verbal instructions upon the formation of a concept, *Journal of General Psychology 6*:400-413.

Eysenck, M. W., & M. T. Keane. (1995/2004) *Cognitive Psychology: A Student's Handbook.* Hove, UK: Erlbaum.

Falkenhainer, B., K. Forbus, & D. Gentner. (1989) The structure mapping engine: algorithm and examples, *Artificial Intelligence 41*:1-63.

Feigenbaum, E. A., & J. Feldman, eds. (1963) *Computers and Thought.* New York: McGraw-Hill.

Feyerabend, P. (1987) *Farewell to Reason.* London: Verso.

Fiedler, A. (2001) *User-adaptive proof explanation*, PhD thesis, Universität des Saarlandes.

Fikes, R. E., & N. Nilsson. (1971) STRIPS: a new approach to the application of theorem proving to problem solving, *Artificial Intelligence 2*:189-208.

Fikes, R. E., P. E. Hart, & N. J. Nilsson. (1972) Learning and Executing Generalized Robot Plans, *Artificial Intelligence 3*:251-288.

Fleuriot, J. D., & L. C. Paulson. (2000) Mechanizing Nonstandard Real Analysis, *London Mathematical Society Journal of Computation and Mathematics 3*:140-190.

Fodor, J. A. (1975) *The Language of Thought.* Cambridge, MA: Harvard University Press.

Fodor, J. & Z. Pylyshyn. (1988) Connectionism and cognitive architecture: a critical analysis, *Cognition 28*:3-71.

Forbus, K. D., D. Gentner, & K. Law. (1995) MAC/FAC: a model of similarity based retrieval, *Cognitive Science* 19:141-205.

Forgy, C. (1982) Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artificial Intelligence 19*:17-37.

Foss, B. M., ed. (1966) *New Horizons in Psychology*. Harmondsworth, England: Penguin.

Fotland, D. (1993) Knowledge Representation in Many Faces of Go. `http://pw1.netcom.com/~fotland/manyfaces.html`

Fox, M., & D. Long. (2001) Hybrid STAN: Identifying and Managing Combinatorial Optimisation Sub-problems in Planning. In *Proceedings of IJCAI-2001*.

Frege, G. (1879). *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Halle: L. Niebert. *trans.* Begriffsschrift: a formula language, modeled upon that of arithmetic, for pure thought. In J. van H., ed. (1967) *From Frege to Goedel: A Source Book in Mathematical Logic, 1879-1931*. Cambridge, MA: Harvard University Press.

Gailly, J-L. (1999) Jean-loup's Go page. `http://w3c.teaser.fr/~jlgailly/go.html`

Geis, M. C., & A. M. Zwicky. (1971) On invited inferences, *Linguistic Enquiry* 2:561-566.

Gentner, D. (1983) Structure-mapping: a theoretical framework for analogy, *Cognitive Science* 7:155-170.

Gentner, D., & R. Landers. (1985) Analogical remindings: A good match is hard to find. In *Proceedings of the International Conference on Systems, Man, and Cybernetics*.

Gentner, D., & A. B. Markman. (1997) Structure-mapping in analogy and similarity, *American Psychologist 52*:45-56.

Gentner, D., & J. Medina. (1998) Similarity and the development of rules, *Cognition 65*:263-297.

Gentner, D., M. J. Ratterman, & K. Forbus. (1993) The roles of similarity in transfer, *Cognitive Psychology 25*:524-575.

Gentzen, G. (1935) Untersuchungen über das logische Schliessen, *Mathematische Zeitschrift 39*:176-221. *trans.* (1964), Investigations into logical deduction, *American Philosophical Quarterly 1*:288-306.

Gick, M. L, & K. J. Holyoak. (1980) Analogical problem solving, *Cognitive Psychology 12*:306-355.

———————. (1983) Schema induction and analogical transfer, *Cognitive Psychology* 15(1):1-38.

Goodman, N. (1972) *Problems and Projects*. New York City: Bobbs-Merrill.

Goldman, D., & D. Homa. (1977) Integrative and metric propoerties of abstracted information as a function of category discriminability, instance variability, and experience. *Journal of Experimental Psychology: Human Learning and Memory 3*:375-385.

Gordon, M. J. C., & T. F. Melham, eds. (1993) *Introduction to HOL: a theorem-proving environment for higher-order logic*. Cambridge: Cambridge University Press.

Gordon, M. J., A. J. Milner, & C. P. Wadsworth. (1979) Edinburgh LCF - A mechanised logic of computation, Lecture Notes in Computer Science, vol. 78. New York: Springer-Verlag.

Goswami, U. (1996) Analogical reasoning and cognitive development, *Advances in Child Development and Behavior 76*:91-136.

Greiner, R. (1988) Learning by understanding analogies, *Artificial Intelligence 35*:81-125.

de Groot, A. D. (1946) *Het Denken van den Schaker*. The Hague: Mouton. *trans.* and revised by A. D. de Groot & G. W. Baylor. (1965) *Thought and Choice in Chess*. The Hague: Mouton.

Grudin, R. (1990) *The Grace of Great Things: Creativity and Innovation*. New York: Ticknor and Fields.

Halford, G. S. (1993) *Chlidren's Understanding: the Development of Mental Models*. Hillsdale, NJ: Erlbaum.

Halford, G. S., J. D. Bain, & M. T. Mayberry. (1998) Induction of relational schemas: common processes in reasoning and complex learning, *Cognitive Psychology* 35:201-245.

Halford, G. S., W. H. Wilson, & S. Phillips. (1998) Processing capacity defined by relational complexity: implications for comparative, developmental, and cognitive psychology, *Behavioral and Brain Sciences* 21:803-864.

Halpern, D. F. (2003) *Thought and knowledge: An introduction to critical thinking*. Mahweh, NJ: Erlbaum.

Hammond, K. (1986) *Case-Based Planning: An Integrated Theory of Planning, Learning, and Memory*, PhD thesis, Yale University.

Harris, A. (2001) *Palm Programming for the Absolute Beginner*. Boston, MA: Thomson Course Technology.

Hartson, W. R., & P. C. Wason. (1983) *The Psychology of Chess*. London: Batsford.

Heneveld, A. (1999). *Schema induction via analogy*, Master's thesis, School of Cognitive Science, University of Edinburgh.

Hobbes, T. (1650) *Humaine Nature*.

Hobbes, T. (1651) *Leviathan*.

Hofstadter, D. (1995) *Fluid Concepts & Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought*. New York: Basic Books.

Hofstadter, D., & M. Mitchell. (1995) The copycat project: A model of mental fluidity and analogy-making. In Hofstadter, D., and the Fluid Analogies Research group, ed. *Fluid Concepts and Creative Analogies*. Basic Books.

Holding, D. H. (1985) *The Psychology of Chess Skill*. Hillsdale, NJ: Erlbaum.

Holyoak, K. J., & B. A. Spellman. (1993) Thinking, *Annual Review of Psychology* 44:265-315.

Holyoak, K. J., & P. Thagard. (1989a) Analogical mapping by constraint satisfaction, *Cognitive Science* 13:295-355.

———————. (1989b) *Mental Leaps: Analogy in Creative Thought*. Cambridge, MA: MIT Press.

Homa, D., & D. Chambliss. (1975) The relative contribution of common and distinctive information on the abstraction from ill-defined categories, *Journal of Experimental Psychology: Human Learning and Memory* 1:351-359.

Howe, J. (1994) Artificial Intelligence at Edinburgh University : a Perspective. `http://www.dai.ed.ac.uk/AI_at_Edinburgh_perspective.html`

Hume, D. (1748) *An Enquiry Concerning Human Understanding*.

Hummel, J. E., & K. J. Holyoak. (1997) Distributed representations of structure: a theory of analogical access and mapping, *Psychological Review* 104:427-466.

IGS (1999) The Internet Go Server. http://igs.joyjoy.net/

Ireland, A., M. Jackson, & G. Reid. (1999) Interactive Proof Critics, *Formal Asp. Comput. 11(3)*: 302-325.

James, W. (1890) *The Principles of Psychology*. http://www.yorku.ca/dept/psych/classics/James/Principles/prin12.htm

Jaskowski, S. (1934) On the Rules of Suppositions in Formal Logic, *Studia Logica 1*.

Johnson, T. R. (1997) Control in Act-R and Soar. In *Proceedings of the 19th Annual Conference of the Cognitive Science Society*, 343-348.

Johnson-Laird, P. N. (1983) *Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness*. Cambridge: Cambridge University Press.

Johnson-Laird, P. N., & R. M. J. Byrne. (1991) *Deduction*. Hillsdale, NJ: Erlbaum.

Johnson-Laird, P. N., & M. J. Steedman. (1978) The psychology of syllogisms, *Cognitive Psychology 10*:64-99.

Johnson-Laird, P. N., & P. C. Wason, eds. (1977) *Thinking: Readings in Cognitive Science*. Cambridge: Cambridge University Press.

Jones, G. (1996) Technical trip report on visits to University of Michigan (Soar) and Carnegie-Mellon University (ACT-R) pertaining to discussions on Computational Modelling. http://www.ehis.navy.mil/gjones.htm

Katz, J. J., & P. M. Postal. (1964) *An Integrated Theory of Linguistic Descriptions*. Cambridge, MA: MIT Press.

Keane, M. T. (1988) *Analogical Problem Solving*. Chichester, UK: Ellis Horwood.

Keane, M. T., T. Ledgeway, & S. Duff. (1994) Constraints on analogical mapping: a comparison of three models, *Cognitive Science 18*:387-438.

Kershaw, T. C., & S. Ohlsson. (2001) Training for Insight: The Case of the Nine-Dot Problem. In *Proceedings of the 23rd Annual Conference of the Cognitive Science Society*.

Knoblich, G., & F. Wartenberg. (1998) Unnoticed hints facilitate representational change in problem solving, *Zeitschrift für Psychologie 206*:207-234.

Koton, P. (1988) a *Using Experience in Learning and Problem Solving*, PhD thesis, Massachusetts Institute of Technology.

Koffka, K. (1935) *Principles of Gestalt Psychology*. New York City: Harcourt, Brace, and World.

Kohler, W. (1925) *The Mentality of Apes*. New York City: Harcourt Brace Jovanovich.

Kolodner, J. (1993) *Case-Based Reasoning*. San Mateo, CA: Morgan Kaufmann.

Komatsu, L. K. (1992) Recent views of conceptual structure, *Psychological Bulletin 112*:500-526.

Kosslyn, S. M. (1994) *Image and Brain: the Resolution of the Imagery Debate*. Cambridge, MA: MIT Press.

Kuehne, S. E., K. D. Forbus, & D. Gentner. (1999) Category Learning as Incremental Abstraction using Structure-Mapping, poster presentation, 21st Annual Meeting of the Cognitive Science Society. Vancouver, BC.

Kuhn, T. (1962) *The Structure of Scientific Revolutions*. Chicago: University of Chicago Press.

Kung, S. Y. (1998) *Digital Neural Networks*. Upple Saddle River, NJ: Prentice-Hall.

Laird, J. E., A. Newell, & P. Rosenbloom. (1987) SOAR: an architecture for general intelligence, *Artificial Intelligence 33*:1-64.

Laird, J. E., P. S. Rosenbloom, & A. Newell. (1986) Chunking in Soar: The Anatomy of a General Learning Mechanism, *Machine Learning 1*:11-46.

Lakoff, G. (1987) *Women, Fire, and Dangerous Things*. Chicago: Chicago University Press.

Landauer, T. K., & S. T. Dumais. (1997) A solution to Plato's problem: The Latent Semantic Analysis theory of acquisition, induction and representation of knowledge, *Psychological Review 104*:211-240. `http://lsa.colorado.edu/papers/plato/plato.annote.html`

Lighthill, J. (1973) Artificial Intelligence: A General Survey. In Artificial Intelligence: a paper symposium, Science Research Council (UK).

Luger, G. F. (2005) *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Reading, MA: Addison Wesley Longman.

Luria, A. R. (1975) *The mind of a mnemonist*. New York: Basic Books.

Macnamara, J., & G. E. Reyes. (1994) *The Logical Foundations of Cognition*. Oxford: Oxford University Press.

Maier, N. R. F. (1930) Reasoning in humans: I. On direction, *Journal of Comparative Psychology 10*:115-143.

Maier, N. R. F. (1931) Reasoning in humans: II. The solution of a problem and its appearance in consciousness, *Journal of Comparative Psychology 12*:181-194.

Manning, A., Ireland, A., and Bundy, A. (1993). Increasing the versatility of heuristic based theorem provers. In Voronkov, A., ed., *International Conference on Logic Programming and Automated Reasoning*, 194-204.

Marling, C. R., G. J. Petot, & L. S. Sterling. (1999) Integrating case-based and rule-based reasoning to meet multiple design constraints, *Computational Intelligence 15*(3):308-332.

Marr, D. (1982) *Vision*. San Francisco: W. H. Freeman.

Mayer, R. E. (1983) *Thinking, Problem Solving, and Cognition*. New York: Freeman.

McCarthy, J. (2000) Review of "Artificial Intelligence: A General Survey". `http://www-formal.stanford.edu/jmc/reviews/lighthill/lighthill.html`

McClelland, J. L. (1981) Retrieving general and specific information from stored knowledge of specifics. In *Proceedings of the 3rd Annual Meeting of the Cognitive Science Society*, 170-172.

McClelland, J. L., D. E. Rumelhart, & the PDP Research Group. (1986) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vols. 1 and 2. Cambridge, MA: MIT Press.

McCloskey, M. E., & S. Glucksberg. (1978) Natural categories: well defined or fuzzy sets?, *Memory & Cognition 6*:462-472.

Medin, D. L., R. L. Goldstone, & D. Gentner. (1993) Respects for similarity, *Psychological Review 100*:254-278.

Medin, D. L., & M. M. Shaffer. (1978) Context theory of classification learning, *Cognitive Psychology 85*:207-238.

Melis, E., *et al.* (1999) "The Omega Project: Analogy-Driven Proof Plan Construction". Project web page at the Arbeitsgruppe Siekman, Universität des Saarlandes. `http://www.ags.uni-sb.de/projects/deduktion/analogy.html`

Melis, E., & J. Whittle. (1999) Analogy in Inductive Theorem Proving, *Journal of Automated Reasoning 22(2)*: 117-147.

Meyer, B. (1988/1997) *Object-oriented Software Construction*. New York: Prentice-Hall International.

Michalewicz, Z., & D. B. Fogel. (2002) *How to Solve It*. New York: Springer.

Michie, D., ed. (1979) *Expert Systems in the Micro-electronic Age*. Edinburgh: Edinburgh University Press.

Mill, J. (1829) *Analysis of the Phenomena of the Human Mind*. London: Baldwin and Cradock.

Mill, J. S. (1843) *A System of Logic*.

Miller, G. A. (1956) The magic number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review 63*:81-93.

Millikan, R. G. (2000) *On Clear and Confused Ideas*. Cambridge: Cambridge University Press.

Milner, R. (1972) *Logic for computable functions; description of a machine implementation*, Technical Report STAN-CS-72-288, AI Memo 169, Stanford University.

Moses, J. (1967) *Symbolic Integration*, MAC-TR-47, Project MAC, MIT.

Muggleton, S., ed. (1992) *Inductive Logic Programming*. San Diego, CA: Academic Press. Golem source code at `http://www.comlab.ox.ac.uk/oucl/groups/machlearn/golem.html`

Müller, B. (1999) Use specifity of cognitive skills: Evidence for production rules?, *Journal of Experimental Psychology: Learning, Memory, and Cognition 25(1)*:191-207.

Murdock, J. W., M. Simina, J. Davies, & G. Shippey. (1998) Modeling Invention by Analogy in ACT-R. In *Proceedings of the 20th Annual Conference of the Cognitive Science Society*, 740-745.

Murphy, G. L. (2002) *The big book of concepts*. Cambridge, MA: MIT Press.

Nadel, L., ed. (2000) *Encyclopedia of Cognitive Science*. London: MacMillan.

Nau, D. S., J. J. Smith, & K. Erol. (1998) Control Strategies in HTN Planning: Theory versus Practice. In *AAAI-98/IAAI-98 Proceedings*, 1127-1133.

Nau, D., Y. Cao, A. Lotem, & H. Muñoz-Avila. (1999) SHOP: Simple Hierarchical Ordered Planner. In *Proceedings of IJCAI-1999*, 968-975.

Neubacher, A. (1992) An introduction to the symbolic integration of elementary functions. Technical Report RISC-Linz Report Series No. 92-66, Research Institute for Symbolic Computation, Johannes Kepler University.

Newell, A. & H. A. Simon. (1956) The logic theory machine, *IRE Transactions on Information Theory IT-2(3)*:61-79.

Newell, A., & H. A. Simon. (1972) *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall.

Newell, A. Rosenbloom, & P. S., Laird, J. E. (1989) Symbolic Architectures for Cognition. In Posner, ed. (2005).

Nipkow, T., L. Paulson, *et al.*. (2006) Isabelle. `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`

Norman, D. A. (1980) Twelve issues for cognitive science, *Cognitive Science 4(1)*:1-32.

Norman, D. A. (1982) *Learning and Memory*. San Francisco: Freeman.

Nosofsky, R. M. (1984) Choice, similarity, and the context theory of classification, *Journal of Experimental Psychology: Learning, Memory, and Cognition 10*:104-114.

Osherson, D. N. (1974) *Logical Abilities in Children*, vol. 2: *Logical Inference: Underlying Operations*. Hillsdale, NJ: Erlbaum.

Ostermiller, S. (2006) Tic-Tac-Toe Strategy. `http://ostermiller.org/tictactoeexpert.html`

Owen, S. (1990) *Analogy for Automated Reasoning*. San Diego: Academin Press.

Paivio, A. (1986) *Mental Representations: A Dual-Coding Approach*. Oxford: Oxford University Press.

Pavlov, I. P. (1928) *Lectures on Conditioned Reflexes*, vol. I. London: Lawrence and Wishart.

Pearl, J. (1984) *Heuristics*. Reading, MA: Addison-Wesley.

Pearl, J. (2000) *Causality: Models, Reasoning, and Inference*. Cambridge: Cambridge University Press.

Piaget, J. (1983) Piaget's theory. In P. Mussen, ed. *Handbook of Child Psychology*, vol. 4. New York: Wiley.

Peirce, C. S. (1885) On the Algebra of Logic: A Contribution to the Philosophy of Notation, *American Journal of Mathematics 7.2*:180-202.

Pirolli, P., & J. R. Anderson. (1985) The role of learning from examples in the acquisition of recursive programming skills, *Canadian Journal of Psychology 39*:240-272.

Ploetzner, R., & K. VanLehn. (1997) The acquisition of qualitative physics knowledge during textbook-based physics training, *Cognition and Instruction 15(2)*.

Plotkin, G. D. (1971) *Automatic Methods of Inductive Inference*, PhD thesis, University of Edinburgh.

Pólya, G. (1945/1957) *How to Solve It*.

Pólya, G. (1957) *Mathematics and Plausible Reasoning*.

Pólya, G. (1962) *Mathematical Discovery*.

Popper, K. (1963) *Conjectures and Refutations: The Growth of Scientific Knowledge*.

Posner, M. I., ed. (1989/2005) *Foundations of Cognitive Science*. Cambridge, MA: MIT Press.

Posner, M. I., & S. W. Keele. (1968) On the genesis of abstract ideas, *Journal of Experimental Psychology 77*:353-363.

Quinlan, J. R. (1993) *C4.5: Programs for Machine Learning*. San Diego, CA: Morgan Kaufman.

Ramscar, M. (1999) *The Relationship between Analogy and Categorisation in Cognition*, PhD thesis, University of Edinburgh.

Ramscar, M., & H. Pain. (1996) Can a Real Distinction Be Drawn between Cognitive Theories of Analogy and Categorisation? In *Proceedings of the 18th Annual Conference of the Cognitive Science Society*, 346-351.

Ramscar, M., H. Pain, & R. Cooper. (1997) Is there a Place for Semantic Similarity in the Analogical Mapping Process? In *Proceedings of the 19th Annual Conference of the Cognitive Science Society*, 632-637.

Ramscar, M., & D. Yarlett. (2000) The use of a high-dimensional, environmental context space to model retrieval in analogy and similarity-based transfer. In *Proceedings of the 23rd Annual Conference of the Cognitive Science Society*.

Rattermann, M. J., & D. Gentner. (1987) Analogy and similarity: Determinants of accessibility and inferential soundness. In Proceedings of the 9th Annual Meeting of the Cognitive Science Society, 23-34.

Reed, S. K. (1972) Pattern recognition and categorization, *Cognitive Psychology 3*:382-407.

Reiss, M. (1999) Mick's Computer Go Page. `http://www.reiss.demon.co.uk/webgo/compgo.htm`

Reitman, J. (1976) Skilled perception in Go: deducing memory structures from inter-response times, *Cognitive Psychology 8*:336-356.

Richardson, J., A. Smaill, A., & I. Green. (1998) System description: proof planning in higher-order logic with LambdaCLAM. In C. Kirchner & H. Kirchner, eds., *Fifteenth International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence*, vol. 1421. Lindau, Germany: Springer-Verlag.

Riesbeck, C. K., & R. C. Schank. (1989) *Inside Case-Based Reasoning*. Hillsdale, NJ: Erlbaum.

Rips, L. J. (1983) Cognitive Processes in propositional reasoning, *Psychological Review 90*:38-71.

———————. (1990) Reasoning, *Annual Review of Psychology 41*:321-353.

———————. (1994) *The Psychology of Proof*. Cambridge, MA: MIT Press.

Risch, R. H. (1969) The Problem of Integration in Finite Terms. *Transactions of the American Mathematical Society 139*:167-189.

Rissland, E. L., & K. D. Ashley. (1987) A case-based system for trade secrets law. In *Proceedings of the 1st International Conference on Artificial Intelligence and Law*.

Robinson, J. A. (1965) A machine-oriented logic based on the resolution principle, *Journal of the ACM 12(1)*:23-49.

Rosch, E., & B. B. Lloyd, eds. (1978) *Cognition and Categorization*. Hillsdale, NJ: Erlbaum.

Rosch, E., & C. B. Mervis. (1975) Family resemblances: studies in the internal structure of categories, *Cognitive Psychology 7*:573-605.

Ross, B. H. (1987) This is like that: the use of earlier problems and the separability of similarity effects, *Journal of Experimental Psychology: Learning, Memory, and Cognition 13*:629-639.

Rubinstein, M. F. (1975) *Patterns of Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall.

Rumelhart, D. E. (1980) Schemata: The building blocks of cognition. In Spiro, R., B. Bruce, & W. Brewer, eds. *Theoretical issues in reading comprehension*, New York: Lawrence Erlbaum.

Russell, S., & P. Norvig. (1995) *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice-Hall.

Ruth, M., & B. Hannon. (1999) Creating Knowledge by Analogy, Working Paper 9914, The Center for Energy and Environmental Studies, Boston University. `http://www.bu.edu/cees/research/workingp/pdfs/9914.pdf`

Sabella, M. S. (1999) *Using the Context of Physics Problem Solving to Evaluate the Coherence of Student Knowledge*, PhD thesis, University of Maryland. `http://www.physics.umd.edu/rgroups/ripe/perg/dissertations/Sabella/`

Saito, Y., & A. Yoshikawa. (1996) Perception in TsumeGo under 4 Seconds Time Pressure. In *Proceedings of the 18th Annual Conference of the Cognitive Science Society*, 868.

Samuel, A. L. (1959) Some studies in machine learning using the game of checkers, *IBM Journal of R & D* 3:211-229.

Schank, R. C. (1982) *Dynamic Memory*. Cambridge: Cambridge University Press.

Schank, R. C. (1986) *Explanation Patterns: Understanding Mechanically and Creatively*. Hillsdale, NJ: Erlbaum.

Schank, R. C., & R. P. Abelson. (1977) *Scripts, Plans, Goals, and Understanding*. Hillsdale, NJ: Erlbaum.

Schank, R. C., A. Kass, & C. K. Riesbeck, eds. (1994) *Inside Case-Based Explanation*. Hillsdale, NJ: Erlbaum.

Schoenfeld, A. H. (1979) Explicit Heuristics Training as a Variable in Problem Solving Performance, *Journal for Research in Mathematics Education 10(3)*:173-187.

Seifert, C. M., K. J. Hammond, H. M. Johnson, T. M. Converse, T. F. McDougal, & S. W. Vanderstoep. (1994) Case-based learning: Predictive features in indexing, *Machine Learning 16(1-2)*:37-56.

Shallice, T., & E. K. Warrington. (1970) Independent functioning of verbal and memory stores: a neuropsychological study, *Quarterly Journal of Experimental Psychology* 22:261-273.

Simoudis, E., & J. S. Miller. (1991) The application of CBR to help desk applications. In *Proceedings of the DARPA Case-Based Reasoning Workshop*.

Slagle, J. R. (1963) A heuristic program that solves symbolic integration problems in freshman calculus, *Journal of the Association for Computing Machinery 10(4)*:507-520.

Skorstad, J., D. Gentner, & D. Medin. (1988) Abstraction Processes during Concept Learning: A Structural View. In *Proceedings of the 10th Annual Conference of the Cognitive Science Society*.

Smith, E. E., E. J. Shoben, & L. J. Rips. (1974) Structure and process in semantic memory: a feature model for semantic decisions, *Psychological Review 81*:214-241.

Smith, S. J., & D. S. Nau. (1996) A planning approach to declarer play in bridge, *Computational Intelligence 12*(1):106-130.

Sobel, D. M., J. B. Tenenbaum, & A. Gopnik. (2004) Children's causal inferences from indirect evidence: Backwards blocking and Bayesian reasoning in preschoolers, *Cognitive Science 28(3)*:303-333. `http://ihd.berkeley.edu/backwards_blocking.pdf`

Sowa, J. F. (1984). *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA: Addison-Wesley.

Steedman, M. (in press) *The Productions of Time*.

Steele, G. L. (1990) *Common Lisp the Language*, 2nd ed. Woburn, MA: Digital Press. `http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html`

Stenning, K., & M. van Lambalgen. (2001) Semantics as a Foundation for Psychology: A Case Study of Wason's Selection Task, *Journal of Logic, Language and Information 10*:273-317.

Stenning, K., & R. Tobin. (1994) Assigning information to modalities: comparing graphical treatments of the syllogism. Mimeo, Human Communication Research Centre, University of Edinburgh.

Stoutamire, D. (1991) Machine Learning, Game Playing, and Go, Technical Report TR 91-128. Cleveland, OH: Case Western Reserve University. `http://www.stoutamire.com/david/publications.html`

van der Steen, J. (1999) Go, an Addictive Game. `http://www.cwi.nl/people/jansteen/go/index.html`

Tesauro, G. (1995) Temporal difference learning and TD-Gammon, *Communications of the ACM 38*(3). `http://www.research.ibm.com/massive/tdl.html`

Tulving, E., & W. Donaldson, eds. (1972) *Organization of Memory*. New York: Academic Press.

Turing, A. (1936) On Computable Numbers, with an Application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society*, series 2, vol. 42.

Tversky, A. (1977) Features of similarity, *Psychological Review 84*:327-352.

Tversky, A., & D. Kahneman. (1973) Availability: a heuristic for judging frequency and probability, *Cognitive Psychology 5*:207-232.

VanLehn, K. (1996) Cognitive skill acquisition. In J. T. Spence, ed., *Annual Review of Psychology*, vol. 47, 513-539.

VanLehn, K., & R. M. Jones. (1993) What mediates the self-explanation effect? Knowledge gaps, schemas or analogies? In *Proceedings of the 15th Annual Conference of the Cognitive Science Society*, 1034-1039.

Veale, T. (1999) The metaphor home page: Sapper. `http://www.compapp.dcu.ie/~tonyv/`

Veloso, M., J. Carbonell, A. Perez, D. Borrajo, E. Fink, & J. Blythe. (1995) Integrating Planning and Learning: The PRODIGY Architecture, *Journal of Theoretical and Experimental Artificial Intelligence 7(1)*.

Warrington, E. K., & L. Weiskrantz. (1968) New methods of testing long-term retention with special reference to amnesic patients, *Nature 217*:972-974.

Wason, P. & D. Shapiro. (1971) Natural and contrived experience in a reasoning problem, *Quarterly Journal of Experimental Psychology 23*:63-71.

Weaver, L. (1999) Computer-Go Mailing List, `http://www.hsc.fr/computer-go/`

Wertheimer, M. (1923) Untersuchen zur Lehre von der Gestalt, II, *Psychologishe Forshung 4*:301-305.

Wertheimer, M. (1924) Über Gestalttheorie. An address before the Kant Society, Berlin, 7th December, 1924. In Ellis, W. D., trans. (1938) *Source Book of Gestalt Psychology*. New York: Harcourt, Brace and Co. `http://www.gestalttheory.net/archive/wert1.html`

Wertheimer, M. (1945/1961) *Productive Thinking*. London: Tavistock Publications.

Wikipedia contributors. (2006) *Wikipedia, The Free Encyclopedia*, individual articles as cited in context or clearly implied, versions dated 1 February 2006 unless otherwise indicated, `http://en.wikipedia.org`

Willmott, S. (1997) *Adversarial Planning Techniques and the Game of Go*, MSc Thesis, University of Edinburgh.

Wilson, R. A., & F. C. Keil. (1999) *The MIT Encyclopedia of the Cognitive Sciences.* Cambridge, MA: MIT Press.

Wilson, S., & J. D. Fleuriot. (2005) Geometry Explorer: Combining Dynamic Geometry, Automated Geometry Theorem Proving and Diagrammatic Proofs. In *Proceedings of the 12th Workshop on Automated Reasoning.* `http://homepages.inf.ed.ac.uk/s0091720/download?file=Wilson2005GeometryExplorer:Combining.pdf`

Winston, P. H. (1982) Learning new principles from precedents and exercises, *Artificial Intelligence 19*:321-350.

Winterstein, D. (2004) *Using Diagrammatic Reasoning for Theorem Proving in a Continuous Domain,* PhD thesis, University of Edinburgh. `http://bigred.homelinux.org/~danielw/academic/thesis/proofReaders.htm`

Wittgenstein, L. (1953) Philosophical Investigations, *trans.* by G. E. M. Anscombe. Oxford: Blackwell.

Yoshikawa, A., & Y. Saito. (1997) A Protocol Study of Problem Solving in Go. In *Proceedings of the 19th Annual Conference of the Cognitive Science Society,* 833.

Zadeh, L. (1965) Fuzzy sets, *Information and Control 8*:338-353.

Zalta, E. N., ed. (2005) The Stanford Encyclopedia of Philosophy. `http://plato.stanford.edu/archives/win2005/entries/aristotle-psychology/`

Zobrist, A. (1970) *Feature Extraction and Representation for Pattern Recognition and the Game of Go,* PhD thesis, University of Wisconsin.